DISTRIBUTED SPATIAL QUERY PROCESSING AND OPTIMIZATION

NAWSHAD FARRUQUE
Bachelor of Science, Islamic University of Technology, 2009

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

I dedicate this thesis to my parents.

Abstract

Applications exist today that require the management of distributed spatial data. Since spatial data is more complex than non-spatial data, performing queries on it requires more local processing (i.e. CPU and I/O) time. Also, due to geographical distribution, data transmission costs must be considered. To reduce these costs, one can employ a distributed spatial semijoin as it eliminates unnecessary objects before their transmission to other sites and the query site.

Most existing work propose different representations of the distributed spatial semijoin between two sites only, with very few works exploring its use for processing a query involving more than two sites. In this thesis, we propose both new approaches for representing the spatial semijoin in a distributed setting, and their use for processing a distributed query consisting of any number of sites. Two strategies are proposed for compactly representing the spatial semijoin that reduce both the data transmission and local processing (CPU+I/O) costs when applied in a distributed spatial query. A Global Encompassing Minimum Bounding Rectangle (GEMBR) is utilized, which is partitioned, mapped and applied in two different ways to approximate the objects in a spatial joining attribute. The first is partition indices, while the second is a bit array representation. Then each spatial semijoin is applied in a multi-site distributed spatial query processing strategy. In addition, the two-site spatial semijoin is extended to handle multiple sites so that we have a benchmark strategy for comparison purposes.

We have tested the query processing algorithms for four sites, which are a part of an actual working distributed system. The algorithms are compared with respect to data transmission cost, CPU time, I/O time and false positive results. The algorithms are superior in many cases at optimizing the above criteria. The bit array representation, which is called Bloom Filter Based Spatial Semijoin (BFSJ), is evaluated with respect to different filter factors and found that the optimized algorithms perform significantly better than the Distributed Naïve Spatial Semijoin strategy when synthetic data was used. Also the Partition and Mapping Based Spatial Semijoin (PMSJ) is 1.38 times faster than BFSJ with respect to processing cost while the BFSJ has a tranmission cost gain of 1.12 over PMSJ. Both algorithms are 18 times faster and have six times less transmission cost than Distributed Naïve Spatial Semijoin (NSPJ). Finally, it is also observed that with the increase of hash functions and filter factor the false positive percentage increases.

Acknowledgments

I am greatly indebted to Dr. Wendy Osborn for all her useful comments, help and the research directions that she provided, through out my two years journey of pursuing Master's (Thesis) degree in Computer Science from the Department of Mathematics and Computer Science at The University of Lethbridge. I am also grateful to my committee members, Dr. Yllias Chali and Dr. Kien Tran, for providing me with their invaluable comments and advice. I would like to express my gratitude to the School of Graduate Studies of the University of Lethbridge for all the financial resources they provided to support this research. I want to thank my friends and all the internet communities, which were beside me while I was implementing my research work. I am greatly moved to see the amount of patience shown and support given by my family during my stay at Canada. In closing, I want to say that, I am so fortunate to be a part of this research, since research is a complete collaboration among the smart minds, which gives ample opportunity to learn, to shape our imagination, to increase our knowledge and to contribute in the greater research community.

Also, we are grateful to WestGrid and Compute Canada- Calcul Canada (www.westgrid.ca) for the use of their resources.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, professionals from different application domains around the world must deal with the management and analysis of spatial data that is geographically distributed. One such application domain is in the area of emergency and disaster management. One example of a distributed disaster management application is for earthquake detection across many Canadian cities at risk, which are geographically distributed [1]. Another example is a distributed emergency management system for Australia [11]. In both cases, it is identified that distributing and storing the associated spatial data locally (as opposed to managing it centrally) will contribute greatly to real-time response in emergency situations. Therefore, spatial data has become an integral part of the world, and storing and querying it has become an important research subject.

Research in distributed spatial query processing and optimization has focused on different distributed spatial data operations and distributed query optimization techniques to reduce the data transmission cost, CPU time and/or I/O time [20]. Approaches for processing distributed relational (e.g.alphanumeric) queries mostly focused on reducing the cost of data transmission, while considering the CPU and I/O time to be negligible [2, 6, 15]. However, due to the complex nature of spatial data, CPU and I/O costs must also be taken into consideration [20].

Most research in distributed spatial query processing explore the optimization of spatial operators, such as the spatial join or semijoin, in a distributed environment. Existing approaches can be grouped into distributed spatial join based approaches [12], distributed spatial semijoin-based approaches on a two-site or simulated multi-site system [13, 22, 14], and distributed Bloom filter approaches [24]. However, very few explore the use of these operators for processing a distributed spatial query that involves more than two sites. Exceptions to this [24, 14] use a simulated distributed environment or a parallel environment for evaluation.

Therefore, we explore new optimizations of the spatial semijoin in a distributed environment, and their use in a multi-site query processing strategy. Two strategies are proposed for compactly representing the spatial semijoin that reduce both the data transmission and local processing (CPU+I/O) costs when applied in a distributed spatial query. They utilize a Global Encompassing Minimum Bounding Rectangle (GEMBR), which is partitioned, mapped and applied in two different ways to approximate the objects in a spatial joining attribute. The first is partition indices, while the second is a Bloom filter [5, 9] representation. Then each spatial semijoin is applied in a multi-site distributed spatial query processing strategy. In addition, the two-site spatial semijoin proposed in [22] is extended for multiple sites so that we have a benchmark strategy for comparison purposes.

We evaluate our query processing strategies in an actual (i.e. not simulated on one machine) distributed system, and show how both approaches outperform the extended spatial semijoin based strategy with respect of processing time and data transmission cost. In addition, the optimized approaches are compared with respect to data transmission cost, processing time and false positive rates.

The remainder of this thesis proceeds as follows. Chapter 2 summarizes works in other areas that are referenced by related work and used by our strategies. Chapter 3 presents the core strategies, which are the GEMBR calculation, partitioning and mapping, that are used by two of the proposed query processing strategies. Chapter 4 presents the strategies for distributed spatial query processing. Chapter 5 presents the performance evaluation of the strategies. Finally, Chapter 6 concludes the thesis and presents future research directions and Chapter 7 (Appendix) includes all the pseudocode of my thesis.

Chapter 2

Background

With recent advances in communication and information technology, there grew a huge necessity for storing large volume of spatial data, and for performing fast and efficient queries on them. For example in a Geographical Information System (GIS), a map is represented using geometric objects, such as points, line and polygons. This chapter will cover a selection of spatial query processing and optimization approaches found in the literature. First, a brief introduction on distributed databases, spatial data, distributed spatial database, spatial query processing and optimization is given. Then some detailed explanation on indexed, non-indexed, parallel and distributed spatial joining techniques are presented. Finally, a summary on limitation of the approach is given.

Ozsu and Valduriez [20] defined a distributed computing system as, "A number of autonomous processing elements, which are not necessarily homogenous, that are interconnected by a computer network and that co-operate in performing their assigned tasks". Therefore, each processing element in a distributed computing system, is a computing device that can execute a program on its own as well as work together with other devices to solve a problem. There are various reasons for using distributed systems, such as [15]:

1. A business is operating its units at different sites.
2. Distributed systems can offer an improved user response time over a centralized system.
3. Many recent technologies are inherently distributed, such as E-commerce websites, news on demand, and medical imaging.
4. A big task can be divided into smaller tasks, and each can be done simultaneously

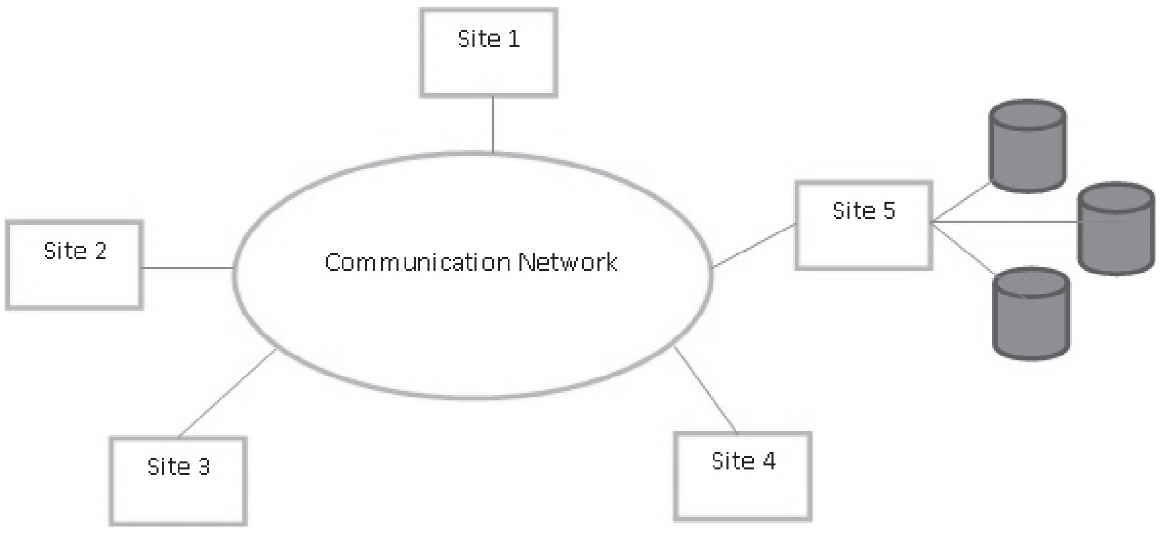


Figure 2.1: Central Database on the Network [15]

by different processors thus processing the whole task. This divide and conquer approach can be used to solve many large computing problems in a time and cost effective manner, because building a huge multiprocessor system is much more costly than building a system having many, simpler processors that are interconnected by a network.

2.1 Distributed Database System

A Distributed Database System (DDBS) is a collection of multiple, logically interrelated databases distributed over a computer network [15]. A distributed database management system (DDBMS) manages the DDBS. Also, it makes the distribution of data transparent to the users. The data in a DDBS are structured and can be accessible through a common interface. The users can interact with the distributed database without having any prior knowledge of the distribution of data. There is a clear distinction between a networked centralized database system and a DDBS. The former consists of different sites that are

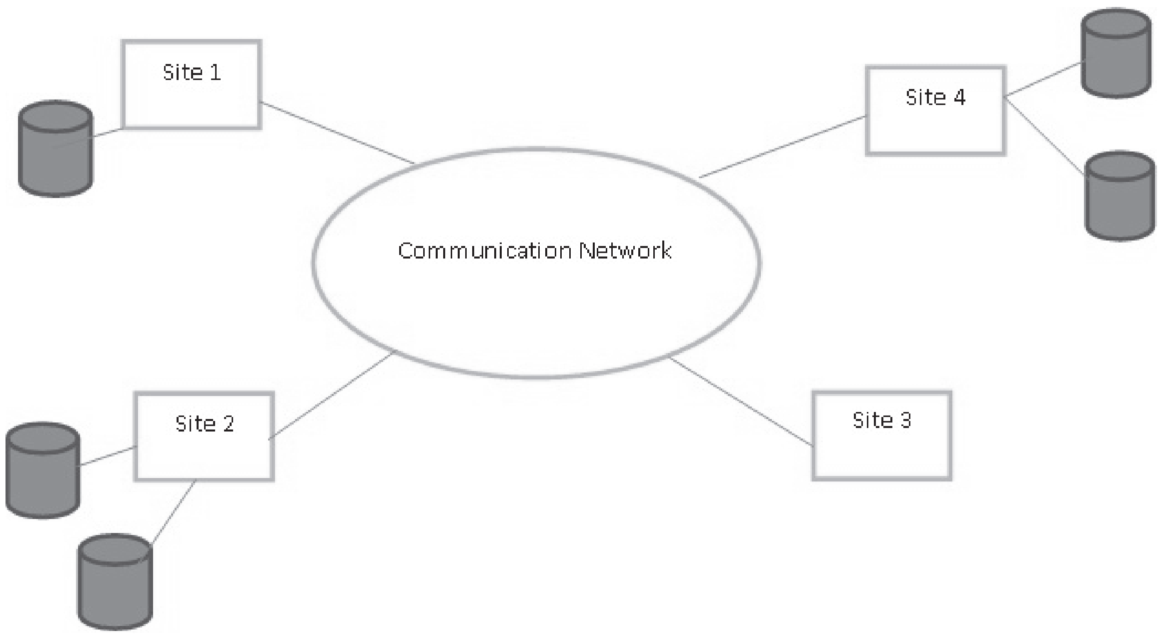


Figure 2.2: DDMS Environment [15]

connected through a network though only one site manages a database which is accessed by all other sites. In the latter, all sites manage a subset of the database. Figure 2.1 depicts a centralized database, which is in site 5 and accessed remotely by other sites 1, 2, 3 and 4. On the other hand, 2.2 depicts the distributed database where each site contains a subset of the entire database.

Depending on how to separate functionality and data representation among different processes, there are three types of DDBS architecture [15]:

1. Peer-to-peer distributed system
2. Client server system
3. Distributed multi-database system (MDBS).

Our work utilizes a form of a peer-to-peer distributed system. This will be summarized next.

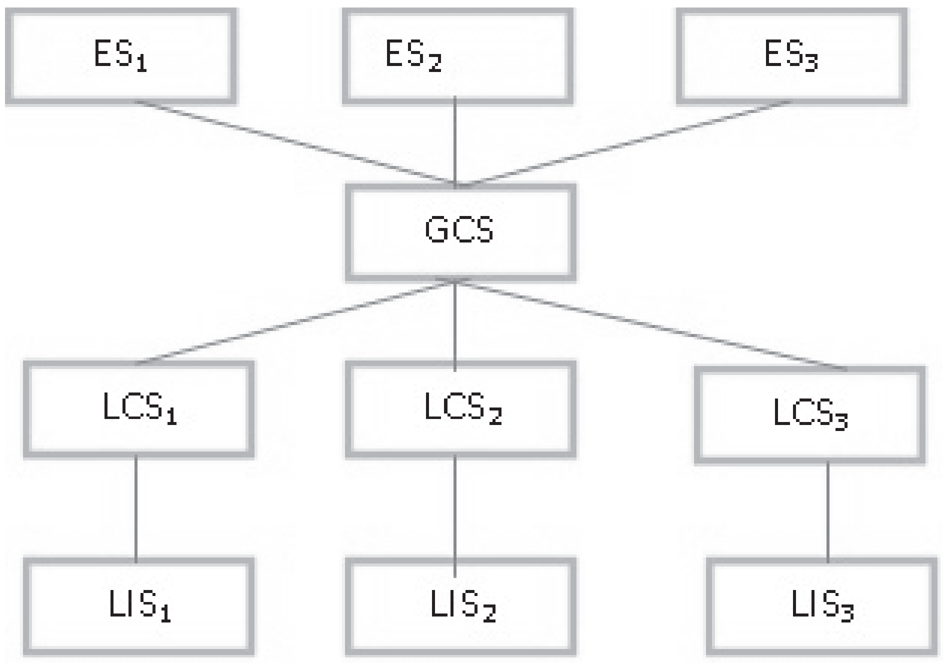


Figure 2.3: Peer-to-peer Database Architecture [15]

In a peer-to-peer distributed system architecture [15], the physical data organization on each machine is different. For this reason, an internal schema definition of each site is needed, which is called a Local Internal Schema (LIS). To handle replication and fragmentation [20], the local organization of data at each site is described using a third layer named a Local Conceptual Schema (LCS). The enterprise view of the data is described by the Global Conceptual Schema (GCS), which is global because it describes the logical structure of data from all sites. The Global Conceptual Schema is the union of all Local Conceptual Schemas. User applications and user accounts are supported by the External Schema (ES), which is defined as the upper layer of the GCS. This model is an extension of the ANSI/SPARC architecture [15], which provides data independence. Location and replication transparency are supported by the LCS and GCS, while network transparency is supported by the GCS. The DDBMS translates a global queries into local queries, each of which are executed by local DDBMS components that communicate with one another. Global mappings are performed by the Global Directory/Dictionary among the local databases and local mapping by the Local Directory/Dictionary among the local database instances. Local database management components are integrated into Global Database by means of global DBMS functions. Local conceptual schemas are mappings of the global schema onto each site. Figure 2.3 depicts such a system.

2.2 Spatial Data

Spatial data [20] is a kind of data that has a specific location in n-dimensional space, and is expressed in respect to a spatial frame of reference. Examples include surface of the earth, a silicon chip and the human body. Point, lines and rectangles are examples of spatial data. Spatial data can be used for representing real life data. For example, spatial data can be used to represent different geographical objects such as rivers (as polygons), buildings (as points) and roads (as lines).

The database that is tailored for storing spatial data is called a spatial database. A spatial database management system is used to manage this data, which provides a better user interface to the data and answers user queries in a very efficient manner.

2.3 Distributed Spatial Database and Query Processing

A distributed spatial database [22] consists of multiple spatial databases that are scattered over a computer network. It is a combination of the idea of a spatial database and distributed database. Distributed spatial query processing involves querying on spatial data which are scattered in distributed servers (i.e. located in different physical locations).

Distributed spatial query processing incurs several costs [20]. CPU costs include the processing needed to calculate the Minimum Bounded Rectangles (MBRs, which is defined in the next section) of the spatial objects and also the geometric operations on them, like intersection, containment and adjacency. More processing needed by the processors indicates more CPU cost. I/O cost generally refers to the number of transfers of a page from/to external memory [19]. Data transmission costs refers the size of the data that is sent to and from the servers.

2.3.1 Spatial Join and Semijoin

Two spatial operators utilized in distributed spatial query processing are the spatial join and spatial semijoin [20]. The spatial join is used to combine two or more spatial datasets with respect to a spatial predicate. The predicate can be a combination of directional distance and topological spatial relations. An example is finding out the intersection between two polygons in a map, where the polygon represents the extent of something (such as disease, natural calamity, etc.). In the case of a non-spatial join, the joining attribute must be of the same type, where for a spatial join, the join attribute can be of different types. Generally, the objects in a spatial attribute are represented by its Minimum Bounded Rectangles (MBR) instead of the objects themselves. A Minimum Bounded Rectangle (MBR) is an approximation of an exact polygon, which is used for reducing the CPU and I/O cost in the filter step (see below) in spatial databases.

There are two steps in spatial joining. These steps are as follows [20]: 1) filter step and 2) refine step. In the filter step, the spatial join is performed with approximations of the spatial joining attributes. The objects that will participate in the join are approximated with their MBRs. It is computationally cheaper to compute intersection or other spatial operations between the query region and the MBR, than with any other arbitrary, irregularly shaped spatial object. It has been seen that if the query region is a rectangle, then at most four computations are needed to determine whether the two rectangles intersect or not

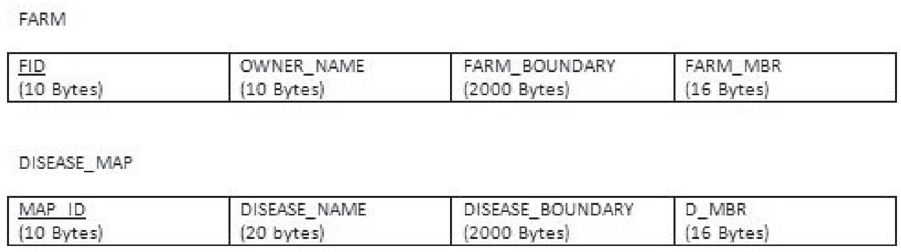
[18]. The filter step results in only the candidates which satisfy the original query. Since some of the candidates may not be the part of the final result, the result of the filter step is processed using exact geometries in the refine step. Though this is a computationally expensive process, the filter step reduces much of its burden.

The spatial semijoin [22] is based on the semijoin [15]. This operator can in many instances reduce the transmission cost of data transfer. The spatial semijoin reduces the transmission cost mainly by:

1. Transferring only the spatial joining attribute and primary key from Site 1 to Site 2,
2. Performing the spatial join of the joining attributes from Site 1 with relation at Site 2 and
3. Transferring only the relevant tuples from Site 2 to Site 1, which are joined with the relation at Site 1.

2.3.2 Example

Distributed spatial query processing including CPU, I/O and transmission cost factors illustrated by the following example summarized from Shekhar and Chawla [20]. Suppose an insurance company is trying to find out the amount of damage that occurred from crop diseases for farmers in a country where all farms have been affected by various crop diseases. Also, suppose that the reimbursement of the insurance company depends on the type of disease that occurred. The insurance company has access to the *FARM* database which is maintained by the country registrars office, and also to the *DISEASE_MAP* database where digital maps of disease spread are maintained by the Department of Agriculture. Instead of obtaining all the necessary data from the two different databases, it can be assumed that the database of the insurance company is a part of a distributed database in which the databases



FARM

FID (10 Bytes)	OWNER_NAME (10 Bytes)	FARM_BOUNDARY (2000 Bytes)	FARM_MBR (16 Bytes)

DISEASE_MAP

MAP_ID (10 Bytes)	DISEASE_NAME (20 bytes)	DISEASE_BOUNDARY (2000 Bytes)	D_MBR (16 Bytes)

Figure 2.4: *FARM* and *DISEASE_MAP* Relation Attributes and Their Size [18]

of the two government agencies reside as well. Figure 2.4 depicts the two relations. The insurance company is looking for an answer to the following query:

SELECT *F.FID*, *D.DISEASE_NAME*

FROM *FARM* F, *DISEASE_MAP* D

WHERE *Intersects*(*F.FARM_BOUNDARY*, *D.DISEASE_BOUNDARY*)

If using as spatial join, the query can be processed in three different ways:

1. The *FARM* relation can be transferred to the site of the *DISEASE_MAP* relation, with the result sent to the insurance company. This will incur a data transmission cost of $(10+10+2000+16) = 2,036,000$ bytes.

2. The *DISEASE_MAP* relation can be transferred to the site of *FARM* relation, with the result sent to the insurance company.

3. Both the *FARM* and *DISEASE_MAP* relation can be sent to the insurance companys database. Although the CPU and I/O costs will not be affected by any of the three choices, the transmission cost will because in each case the query plan is the same but the physical location of the respective databases is not the same.

If a spatial semijoin is used, then the query can be processed in the following way:

1. FID and *FARM_MBR* of the relation *FARM* are projected out and transferred to the site of *DISEASE_MAP*. The number of byte transferred is (10+16)*1000=26,000

Bytes. There are a total 1000 tuples in the *FARM* relation.

2. The *FARM* relation is joined with the *DISEASE_MAP* relation on *FARM_MBR* and *D_MBR* attribute. It is assumed that 10 tuples of *DISEASE_MAP* are selected by this spatial join operation. All 10 tuples are then transferred back to the site containing the *FARM* relation. A total of $(10+20+2000+16)*10 = 20,460$ Bytes are transferred.

3. Join the 10 *DISEASE_MAP* tuples with the *FARM* Relation on the *FARM* relation site. It is assumed that all the farms are affected by some diseases. All 1000 tuples consisting of FID, $OWNER\_NAME$, $DISEASE\_NAME$ are sent to the insurance companys site which results in transferring $(10+10+20)*1000 = 40000$ bytes.

   As seen earlier when processing the query using the spatial join, $2,036,000$ bytes are sent instead of only $26,600+20,460 = 46,460$ bytes, which is almost 90% less.

2.3.3 Bloom Filter

The Bloom filter [5] is an array of *m* bits which can be used to compactly represent a set of *n* items and used for membership queries. In the context of query processing, a Bloom filter of *m* bits can be used to represent a set of *n* distinct attribute values. Given all *m* bits initially set to 0, for each attribute value in *S*, the Bloom filter uses *k* independent hash functions, each with the range $\{1,2,....,m\}$ to produce addresses for *k* Bloom filter locations and to set the bit at each address to 1. To check if an attribute value *x* is member of set *S* then *x* is sent to the same hash functions to re-produce the addresses. If the bits at the reproduced addresses are set to 1, the *x* is a potential member of *S*. For example, in the Figure 2.5, we can see that hash function *h* maps the two members of set S, *a* and *b*, to bloom filter indices 1,7 and 4,8 respectively(i.e. the corresponding bits are set to 1). Now, if we want to test whether value, *t* is the member of the set, it is passed through the same hash function.

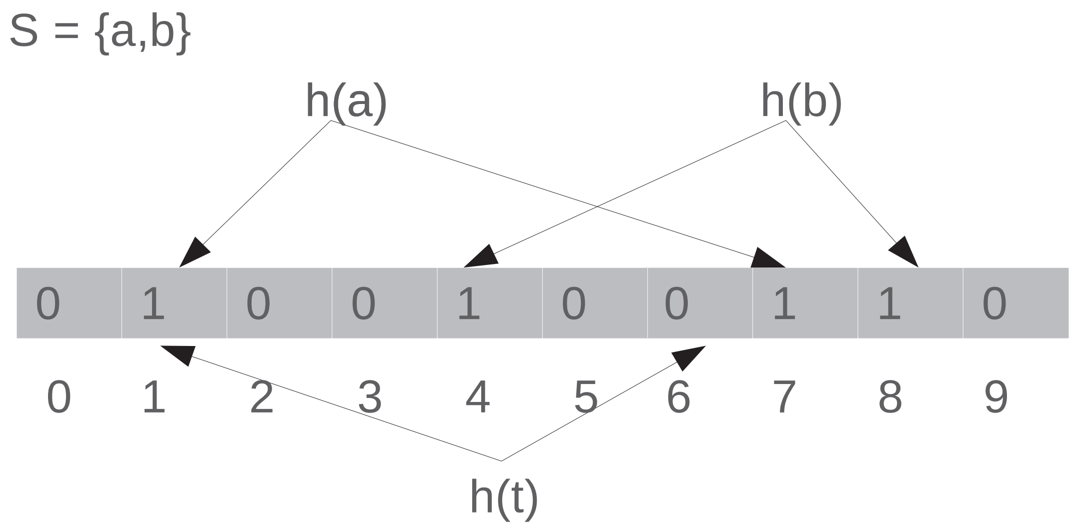


Figure 2.5: Bloom Filter Example

Suppose the hash function calculates for t the indices 1 and 6, since index 6 is set to 0, *t* is not the member of set S.

However, because hash functions produce collisions, there is a certain probability of *false positives* occurring, which means that an attribute value can pass the Bloom filter test and not actually exist in *S*.

The *false positive rate* is quantified in [9] as the following. Given that hash functions are random and all attribute values in *S* have been processed, the probability of a bit being still 0 is:

$$p_{zero} = \left(1 - \frac{1}{m}\right)^{kn} \approx 1 - e^{-\frac{kn}{m}}$$

Hence, the probability of *false positives* occurring is [9]:

$$p_{error} = (1 - p_{zero}) = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

It is found that p_{error} is minimum [9] when

$$k = \frac{m}{n} \ln 2$$

So in our Bloom Filter Based Spatial Semi-join algorithm (BFSJ, see Chapter 4) to get the least possible false positive we have assumed the bloom bit factor or $\frac{m}{n}$ is 1.5 and hence the number of hash functions required is approximately 1.

2.4 Spatial Query Processing Strategies

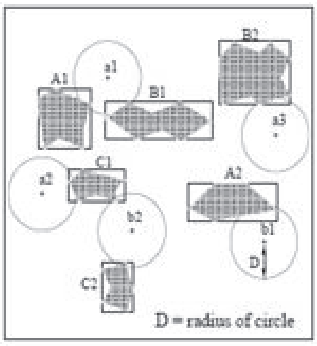
There are many centralized and distributed spatial joining algorithms, which can be grouped as indexed based, non-indexed based, parallel, and distributed spatial semijoin. A selection of the state-of-the-art algorithm from each of these types that are found in the literature are summarized below.

2.4.1 Index Based Approaches

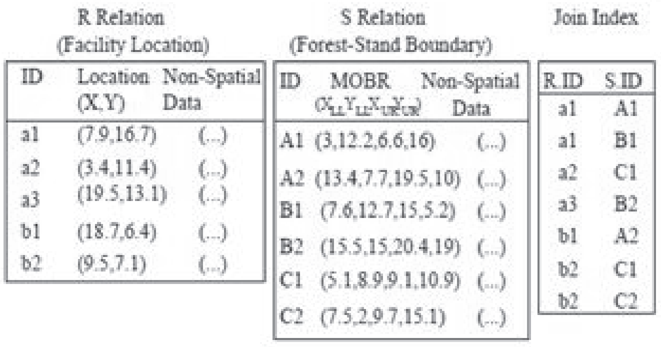
Index based spatial joining means joining two relations, where either or both of them has an index on their joining attribute.

Join Index Based Spatial Join

Shekhar *et al.* [18] proposed a graph partition based approach for optimizing the index based spatial join. They mainly worked on the join index data structure. Pages are grouped using global clustering methods so that the number of redundant page accesses is reduced. The authors stated that previously proposed heuristics for global clustering generally group



(a)Spatial Attribute of R and S


R Relation (Facility Location)

ID	Location (X,Y)	Non-Spatial Data
a1	(7.9,16.7)	(...)
a2	(3.4,11.4)	(...)
a3	(19.5,13.1)	(...)
b1	(18.7,6.4)	(...)
b2	(9.5,7.1)	(...)

S Relation (Forest-Stand Boundary)

ID	MOBR $(X_{LL},Y_{LL},X_{UR},Y_{UR})$	Non-Spatial Data
A1	(3,12.2,6.6,16)	(...)
A2	(13.4,7.7,19.5,10)	(...)
B1	(7.6,12.7,15,5.2)	(...)
B2	(15.5,15,20.4,19)	(...)
C1	(5.1,8.9,9.1,10.9)	(...)
C2	(7.5,2,9.7,15.1)	(...)

Join Index

R.ID	S.ID
a1	A1
a1	B1
a2	C1
a3	B2
b1	A2
b2	C1
b2	C2

(b)R and S relation Table and Join Index

Figure 2.6: Demonstrating the Two Relations and Their Join Index [18]

pages of a single table by using either global sorting or incremental clustering method. Therefore, they introduced two new heuristic approaches: 1) global clustering of group pages in both tables and 2) global clustering of the pages of a single table using join index information. For clustering, the authors used a graph partitioning approach.

A join index is a data structure which is used to store partially materialized relationships in order to speed-up online query processing. It is a bipartite graph of the pairs of surrogates, where each pair of surrogates identifies tuples that participate in a join. For an example, relations *R* and *S* are joined on attribute *R.A*and *S.B*. The join index is defined as J1=(r_i,s_j)|F(r_iA,s_jB), where F is the join predicate and r_i and s_j are the surrogates from R and S respectively.

Figure 2.6 illustrates an example join index of the Facility and Forest Stand relationships, where the join index contains the tuple IDs which matches with the join predicate. This join index is further described by a bipartite graph, G= $\{V_1,V_2,E\}$, where V_1 contains the tuple IDs of relation *R*, V_2 contains the tuple IDs of relation *S* and edge set *E* contains an edge(v_r,v_s) for every pair of surrogates in the join index. Similarly the authors proposed a Page Connectivity Graph(PCG) as B_g = $\{V_1,V_2,E\}$, where V_1 and V_2 are the set of pages from relation *R* and relation *S* respectively and edge E represents the connectivity in which there is at least one pair of surrogates between the two pages.

Using their join index, the authors define the optimization problem as:

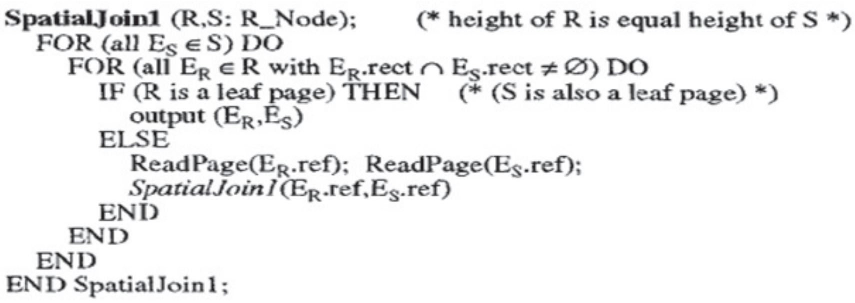
“Given a page connectivity graph PCG = (V,E), representing the join index and a buffer size of $B \leq V$, find out a page access sequence, to minimize the number of page accesses, such that the pages in buffer is not more than B” [18].

To solve this problem the authors present two types of Heuristic methods namely Asymmetric Graph Partitioning approach (AGP) and Symmetric Graph Partitioning (SGP) approach. AGP clusters pages of relation R based on their interaction with the pages of relation S. Redundant I/O of page p of relation S can be reduced if many pages of R are edge connected with p. It can be kept in memory, when reading p first time. The SGP uses a min cut node partitioning approach to cluster the nodes from both relations from their page connectivity graph. A min cut node partition of a graph G=(V,E) is a partition of the node set V into disjoint subsets, while minimizing the number of edges whose end nodes are in different partitions. The CPU cost is fixed since the relations size does not change. The I/O cost depends on the sequence of the access of pages. The PCG is used to determine a schedule of optimal page accesses.

In their paper the authors showed that AGP and SGP out performed existing sort-based and graph-based heuristics. They used the Sequoia [21] data sets for evaluating their strategies. They successfully showed how their algorithms work to resolve the optimization problem by finding the sequence of page accesses that minimize redundant I/O access.

Spatial joins using R-tree

Brinkhoff *et al.* [7] proposed efficient spatial join techniques that use the R-tree [10], and particularly the R*-tree [4]. They presented an efficient variant of R*-tree algorithm which has reduced CPU and I/O.



```
SpatialJoin1 (R,S: R_Node);        (* height of R is equal height of S *)
  FOR (all $E_S \in S$) DO
    FOR (all $E_R \in R$ with $E_R$.rect $\cap$ $E_S$.rect $\neq \emptyset$) DO
      IF (R is a leaf page) THEN     (* (S is also a leaf page) *)
        output ($E_R$,$E_S$)
      ELSE
        ReadPage($E_R$.ref); ReadPage($E_S$.ref);
        SpatialJoin1($E_R$.ref,$E_S$.ref)
      END
    END
  END
END SpatialJoin1;
```

Figure 2.7: SpatialJoin1 Algorithm [7]

The R*-tree [4] is the efficient variant of the R-tree [10] which has more sophisticated insertion and deletion methods. It uses a B+ tree like structure used for hierarchically storing multidimensional objects (or MBRs) and the regions of space that contain them. In an R-Tree, a non-leaf node consists of entries of (ref,rect) where ***ref*** refers to a subtree and ***rect*** is the MBR that contains all rectangles in the subtree. In leaf node, ***ref*** refers to the spatial object record in the database and ***rect*** is the MBR for the spatial object. A common feature of the R-tree is as follows: because rectangles of different nodes can overlap the same region of space, a search may traverse multiple paths which results in bad query performance.

The authors proposed several spatial join algorithms, named as SpatialJoin1 (SJ1), SpatialJoin2 (SJ2), SpatialJoin3 (SJ3), SpatialJoin4 (SJ4), SpatialJoin5 (SJ5). The description of each is given below.

SpatialJoin1(SJ1). Given two R*-trees of equal height *R* and *S*, the *SJ*1 algorithm compares all pairs of rectangles from *R* and *S* for overlap. The algorithm traverses both trees from the top down in a recursive manner. If a pair of rectangles do not overlap, then none in their respective sub trees will overlap. The algorithm is depicted in the Figure 2.7.

To minimize I/O and increase main memory operations, the authors proposed the use of two buffers:

1. the path buffer, which consists of the nodes on a path which was last accessed and
2. the least recent used (LRU) buffer, used for the single nodes instead of a path of nodes.

The authors performed experiments on $SJ1$ for different LRU buffer and page sizes. They found that the best overall performance with respect to execution time is achieved when the page size is 1-2 KB and the buffer size is 0-512 KB. They also found that the spatial join is slightly I/O bound for the page size of 1KB, and becomes more CPU bound with the increase in page size. Therefore, the authors considered a number of approaches for improving the CPU and I/O costs.

SpatialJoin2(SJ2). In the $SJ2$ algorithm, the authors focus on reducing CPU costs by restricting the search space. They achieved this by reducing the number of floating point comparisons that occur between each pair of selected nodes. They only considered two non-leaf node entries E_R and E_S that fulfill the condition $E_R.rect \cap E_S.rect \neq \phi$. Then, the $SJ1$ algorithm is invoked using E_R.ref and E_S.ref as input.

A performance comparison between $SJ1$ and $SJ2$ showed that the number of comparison between them decreases super linearly with the increase of the page size.

SpatialJoin3(SJ3). In the $SJ3$ algorithm, the entries of both R-trees are sorted according to their spatial location in the data space. Then, the plane sweep algorithm [3] is applied to compute the desired pairs of the intersecting entries. The authors found that sorting the entries as soon as they are fetched in the buffer can give better CPU time in both cases when search space is restricted and not restricted.

SpatialJoin4(SJ4). To solve the problem of increased I/O by reading the same page more than one time, which occurred in $SJ3$ algorithm, the $SJ4$ algorithm uses local plane sweep order with pinning. First a pair of entries from indexes R and S is determined using a local plane sweep algorithm [20]. Once the processing of the corresponding sub-trees

E_R.ref and E_S.ref is done, the degree of the rectangles for both of the entries is computed. The degree of a rectangle from R is the number of intersections between the rectangles of R with all the unprocessed rectangles of S. Then the pages with the highest degree are pinned into the buffer. Lastly the join is done between those pinned pages with all other pages.

SpatialJoin5(SJ5). $SJ5$ reduces I/O by using Z-order [16] spatial sorting of node entries. The basic idea behind Z-order is to decompose the total data space into cells of equal size and provide an ordering on this set of cells. First, the intersection between the rectangles of one node and the rectangles of all other nodes is computed. Then, the resulting rectangles are Z-order sorted according to the spatial location of their centers.

The authors found the following in their performance evaluation. The local plane sweep order approach used in $SJ3$ has more performance gain than pinning of pages in $SJ4$ when the buffer is small. With the buffer size of 512KB, the I/O performance is nearly the same. Because the CPU performance of both approaches is the same, $SJ4$ is found to be clearly better than $SJ3$. For a smaller size of buffer $SJ5$ is slightly better than $SJ4$, for larger buffer it is vice versa. But, local Z-order calculation needs CPU time which is not compensated with the little I/O gain. $SJ4$ needs 45% less disk accesses than $SJ1$. With the smaller buffer by a factor 3 to 6, $SJ4$ achieves the same I/O performance as $SJ1$. Finally, the $SJ4$ seemed to be the most efficient algorithm with respect to performance gain because it has improved performance gain than $SJ1$ and $SJ3$.Its I/O performance is almost the same as $SJ5$, though the preprocessing step for $SJ5$ is higher than $SJ4$.

The authors successfully showed the basic algorithm and its enhancement through I/O and CPU time tuning using various heuristics. In the future they are hoping to exploit spatial joins using parallel R*-tree in parallel machines.

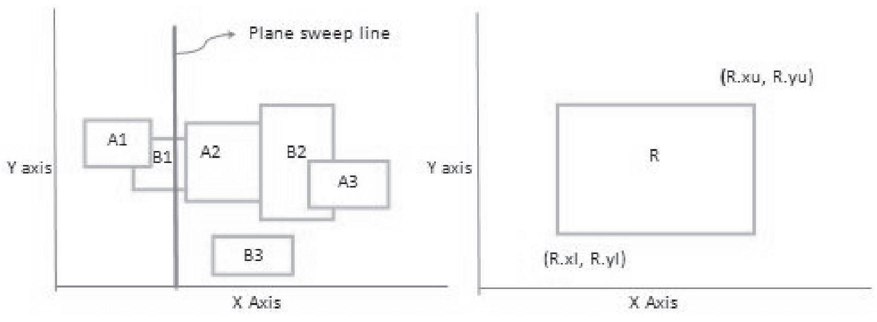


Figure 2.8: Depicting the Plane Sweep Algorithm(left), the Rectangle Co-ordinates to be Used(right) [3]

A1	B1	A2	B3	B2	A3

Figure 2.9: Array of Sorted Rectangles in $A \cup B$ After Sorting

2.4.2 Non-Indexed Based Approaches for Spatial Join

Non-indexed based spatial joining means joining two relations, where neither of them has an index on its joining attribute.

Plane Sweep Approach to Spatial Join

In [20], a plane sweep approach is proposed for determining spatial join. For example, given two sets of MBRs named A and B, where A={A1,A2,A3} and B={B1,B2,B3}, each rectangle is defined by its lower-left-corner (R.xl,R.yl) and upper-right-corner (R.xu,R.yu). This is depicted in the right side image of 2.8. All the rectangles in A and B are sorted according to their lower-left-corners and the following sorted array $A \cup B$, depicted in 2.9, can be found.

The plane sweeping algorithm is applied as follows [20]:

1. The sweep line, which is perpendicular to the x axis, is moved from left to right and stopped at the first entry of $A \cup B$. This is the rectangle A with the smallest R.xl value.

In this example (see Figure 2.8) it is A1.

2. The sorted list of rectangles of B is traversed until the first rectangle of B having $B.xl > A1.xu$ is found. Here, it is $B3$. So at this point the intersection set between [A1.xl, A1.xu] and [B1.xl, B1.xu] will be non empty which implies $A1$ and $B1$ can be a candidate set for overlap.

3. Next, the intersection set between [A1.yl, A1.yu] and [B1.yl, B1.yu] is checked. If it is found non-empty, then A1 and B1 do overlapped. So (A1, B1) is added to the join result. A1 is removed from the set $A \cup B$, because there is no possibility of it being a part of any future overlaps.

4. The sweep line is moved further across the $A \cup B$ set until it reaches the next rectangle entry. This is rectangle $B1$ in the example.

5. Process (2) and (3) is repeated. When $A \cup B$ is empty, the process is stopped.

The plane sweep approach results in (A1, B1), (A2, B1), (A2, B2), and (A3, B2) as candidate pairs for the refinement step. In the refinement step these pairs are checked against the exact spatial objects. Some pairs may be eliminated at this stage if exact geometry computation results that there is no overlap.

Partition Based Spatial Merge Join

Patel and Dewitt [17] proposed the PBSM algorithm, which partitions the relations into disjoint subsets and joins them using a plane sweeping technique [20]. They also presented an empirical performance comparison between the PBSM algorithm, the index based nested loop join and the $SJ1$ algorithm [7].

The PBSM algorithm implements the filter and refinement step in the following manner.

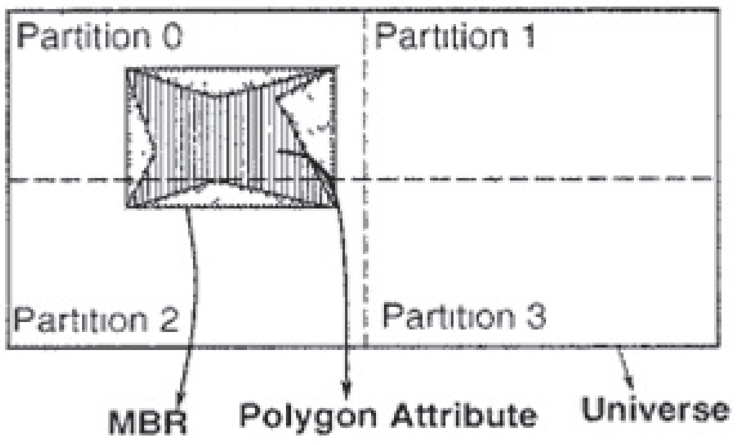


Figure 2.10: Universe and Partitions [17]

For the filter step, two relations -R^{kp}, S^{kp}- are formed, the superscript kp means key-pointer, which contain (MBR, OID) pair from relation R and S, where OID is a unique identifier for each of the spatial objects. Then, if both R^{kp} and S^{kp} fit in main memory, a plane sweeping technique is used to find all pairs of R^{kp} and S^{kp} records that have overlapping MBRs. Finally, for those matching pairs, the OID information is extracted, and this OID pair is the final output.

If R^{kp} and S^{kp} are too big to fit in main memory, then the universe that covers relations R and S is divided into disjoint partitions, and all (MBR, OID) pairs in R^{kp}, S^{kp} are mapped to the partitions that the MBR overlaps.

Figure 2.10 shows an example of an MBR that overlaps both partitions 0 and 2. So the (MBR, OID) pair is inserted into both partitions. After *R* and *S* are partitioned, the algorithm joins the partitions using plane sweeping technique on each partition pair of R^{kp} and S^{kp}.

Since the partitioning in the filter step may insert a tuple into multiple partitions, there could be duplicates in the joined relation. The refinement step eliminates these duplicate (MBR, OID) pairs and examines the actual *R* and *S* tuples from the relations to determine if the actual objects satisfy the joining condition.

A performance comparison among PBSM, indexed nested loop join and *SJ*1 [7] showed that PBSM is more efficient than the other algorithms when either of the relations has no index on it or an index exists on the smaller input. The algorithm *SJ*1 [7] has better performance when the larger input has an index on it, or if both the inputs have a pre-existed index. The authors expect that they will be able to implement their PBSM algorithm in a parallel environment where de-clustering spatial data is a concern.

SSSJ Algorithm

Arge *et al.* [3] have proposed a spatial join algorithm called SSSJ that combines a distribution sweeping technique and main memory plane sweeping. They compared their algorithm with the PBSM algorithm and found that their algorithm is more efficient than the typical PBSM algorithm. In the SSSJ algorithm an initial sorting on the spatial objects is done on the vertical axis. Then, a distribution sweeping technique is applied to partition the input into some vertical strips so that each partition fits into main memory. Then, each partition is brought into main memory for further processing by the algorithm. In this algorithm, partitioning is done along only one axis, which is one enhancement over other similar algorithms. Another enhancement according to the authors is, replication does not occur in their algorithm. Their experiment showed that the SSSJ algorithm performs 25% faster than a typical implementation of PBSM [17], though 10% slower than the improved version of PBSM.

According to the authors, if the data is well behaved [3] then the SSSJ Algorithm gives better performance than any other algorithms. If not, then the PBSM algorithm performs equivalent to the simple sweeping algorithm as it is susceptible to skewed data. In this case, SSSJ achieves optimal worst case bound.

Planning for Distributed Spatial Query Optimization Using Filters

Tripathy *et al.* [23] mainly focused on a tree-based representation and optimization of a query execution plan, called the Spatial Object Algebra (SOA) tree. They proposed an architecture of a query optimizer where the optimized SOA tree is generated by the Object Query Language (OQL) parser from the Structured Query Language (SQL) statement and passed to query optimizer for optimization.

Demonstrating by using an example mixed query which consists of several types of queries, such as containment, adjacency, range and intersection, the authors derived several strategies of optimizing the SOA tree for their query, where they mainly focused on pushing down the operations which are costly, pushing up the operations which are important for join processing in a distributed environment, and the separation of filter and refinement steps in a spatial join.

2.4.3 Parallel Processing Based Approach for Spatial Join

Parallel processing involves partitioning a process into multiple parts and allocating them to different processors, each of which can process the parts independently and thus complete the whole process. In a parallel approach, the node of the two R-trees taking part in a spatial join are processed in a parallel fashion to improve computation time.

Parallel processing of spatial join using R-trees

Brinkhoff *et al.* [8] proposed an R-tree based parallel spatial join algorithm for a parallel platform with shared virtual memory. This algorithm has three main phases: task creation, task assignment and parallel task execution.

To reduce the CPU and I/O time, the three phases are designed so that the spatial locality of the spatial objects is preserved. Dynamic load balancing is maintained by further partitioning the tasks and assigning them to the idle processors. The authors investigated the parallelism of the join algorithm for two main reasons [8]:

1. The sequential join algorithm cannot provide better response time than parallel ones in a multi user environment.

2. Spatial join processing works on a large amount of data and therefore incurs huge computational cost.

The authors identify most important design concern of this algorithm as how to distribute tasks in different processors.

First Approach. The authors first proposed a join algorithm which has no synchronization and communication cost, which focused on reducing the CPU and I/O costs for it.

First, a set of tasks are created. Each task involves performing the sequential algorithm on a pair of sub trees. This phase is performed sequentially in one processor. Second, each task is assigned to a processor, which joins the sub-trees of its task independently from the other processors without any communication taking place among them.

Tasks are assigned to processors by using a local plane sweep order technique. The number of the intersecting *MBR*s of the root of the participating R*-tree is *m* and *n* is the number of processors. The authors assume that *m* is greater than *n*. If not, then the *MBR*s

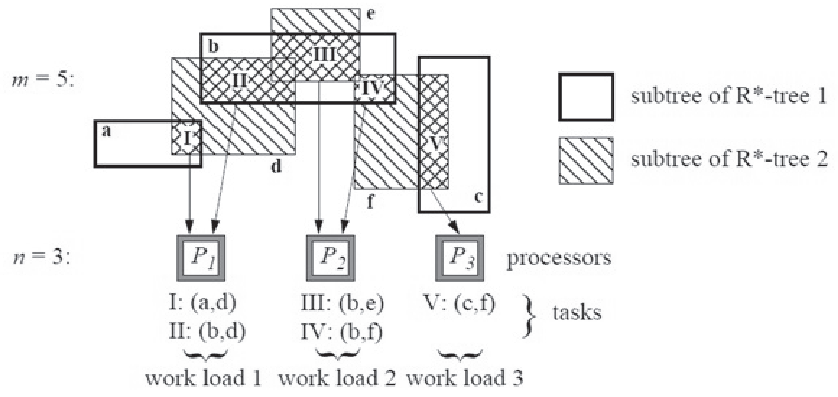


Figure 2.11: Task Assignment of Different Processors [8]

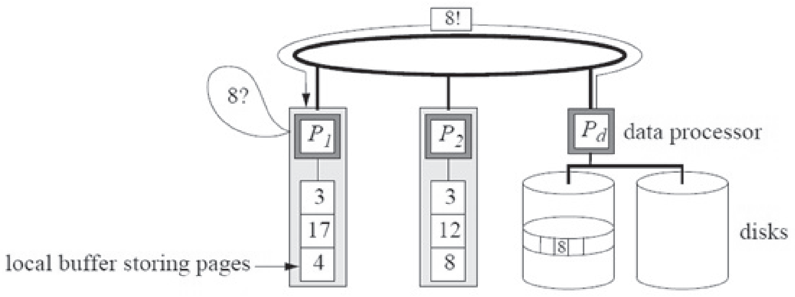


Figure 2.12: Local Buffer Only [8]

that belong to the next lower level nodes are considered. First, tasks are identified using the local plane sweep order. Next, the first m%n number of processors receive $\lceil \frac{m}{n} \rceil$ pairs of subtrees and the rest receive $\lfloor \frac{m}{n} \rfloor$ pairs of subtrees according to the local plane sweep order. Figure 2.11 illustrates the total design, where m=5 and n=3.

The first approach has the following limitations:

1. As the processors do not communicate with each other, there may be situations when same sub-tree needs to be fetched. For example in Figure 2.11, $P1$ and $P2$ are assigned with same sub-tree b. This results in a higher I/O cost.

2. Each task of the workload may not have same execution time, which causes imbal-

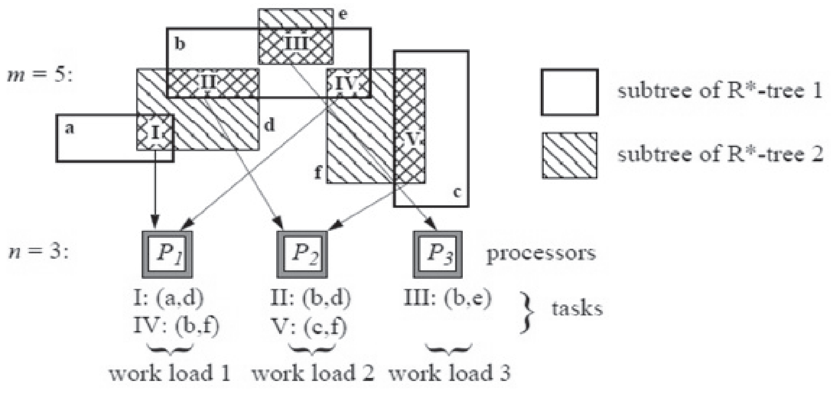


Figure 2.13: Example for Static Round-robin Assignment [8]

anced workloads.

Second Approach. The authors improved their first algorithm by adding synchronization and communication with reduced CPU and I/O cost. To address the issue of multiple fetches of the same subtree, the authors add a global buffer. The global buffer consists of the contents of all local buffers. The access to the global buffer is managed by a Shared Virtual Memory (SVM) manager. The main advantage of it is avoiding an extra look up in the secondary storage, which results in reduced I/O cost.

Task assignment works as follows. First, the *m* intersecting pairs of MBRs are identified using the local plane sweeping technique [3]. Then instead of assigning adjacent sub-trees in one processor, they are assigned in a processor in a round robin fashion using the local plane sweep order. An example is depicted in Figure 2.13. In order to distribute tasks more evenly, the authors proposed a technique where the last two consecutive tasks are kept in a queue called a task queue without processing them. As soon as any processor has finished executing their task, one from the task queue is assigned to it.

So the authors also proposed a task reassignment system where as soon as a processor finishes its task and there is no other task in the task queue, it offers help to the other processors that are still working. A working processor divides its work load in two parts. One part is done by the processor itself, and the other part is re-assigned to the idle processor(s). In order to choose a processor to help, each active processor reports the number of non-processed tasks. The processor having the highest number of non-processed task is chosen as the buddy processor for an idle processor.

The authors evaluated their strategies using a multiprocessor machine having 24 processors. They achieved a linear speed up close to n, where n is the number of disks where the data is stored. The speed up was 22.6 for 24 processors. In the future they hope to investigate distributed spatial join processing using a shared-nothing architecture.

BR-tree

Hua *et al.* [24] proposed a new spatial index structure and a general parallel spatial query processing strategy that uses it. The BR-tree is an R-tree, which includes Bloom filters on every node that handle exact-match object queries. The leaf-node Bloom filters are created from the leaf node objects, while the non-leaf-level filters are created from its MBRs.

Given a BR-tree at every site in the distributed spatial database, their approach is to distribute replicas of all BR-tree root nodes at all sites. Any MBR which is qualified by a root node is transmitted to the BR-tree that contains the original root node for further processing. An evaluation of the BR-tree query processing strategy showed that in terms of query accuracy, average latency, message overhead and memory space, it outperformed other existing structures.

2.4.4 Distributed Spatial Semijoins

Original Approach

Tan *et al.* [22] proposed a spatial semijoin algorithm for use in the distributed spatial database.

The authors state that the transmission cost of a large spatial object is high, but its processing cost is significantly higher. Therefore, the distributed spatial semijoin algorithm considers both the transmission and local processing costs when used for processing a distributed spatial query. *R* and *S* are two spatial relations residing in two different sites, R_{site} and S_{site} respectively. The result of the join is produced at site S_{site}. The spatial semijoin needs to be enhanced to eliminate the following anomalies found in spatial databases [22]:

1. A typical semijoin analyses the distinct values of the joining attribute of *R* and *S* for reducing transmission cost from one site to another. For a large spatial database where there are a huge number of spatial objects even this process incurs high transmission cost. Also, it is likely that fewer duplicates among objects will exist.

2. Performing different operations on the spatial relations such as intersection, containment adjacency is more expensive than comparing two numbers or strings.

They proposed that the MBRs of the spatial objects can be used which conserves the relationship between objects, and simplifies spatial operations. They also proposed a mapping function that maps the distributed spatial semijoin operator to a weaker relationship operator. For example, if $e1$ and $e2$ are two polygons which intersect each other, then their respective approximations, which are two bounding rectangles also intersect each other. This thing is illustrated in Figure 2.14. As it is more convenient to calculate the intersection between two rectangles than between two polygons, sending only the lower left and

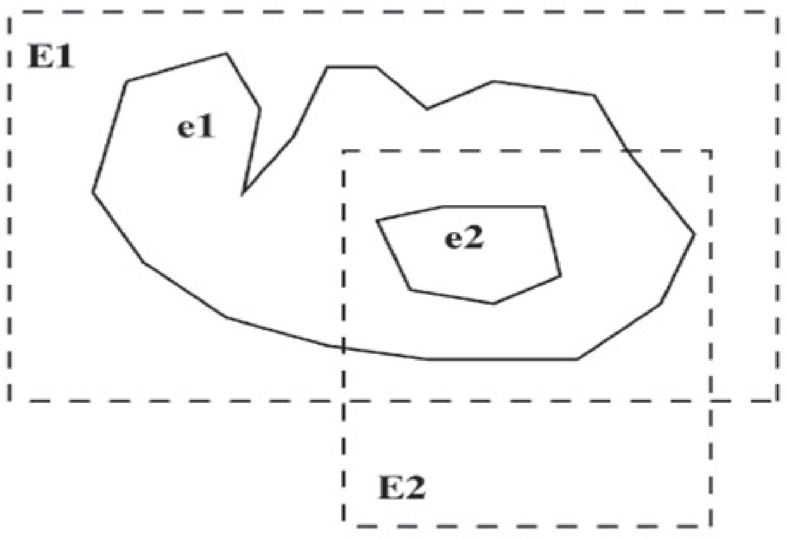


Figure 2.14: E1 Overlaps E2, e1 Includes e2 [8]

upper right vertices of those rectangles is lower in transmission cost than sending many vertices of the polygons.

Using the weaker relationship, the resultant query consists of all records from *R* that may participate in the final result. Since the weaker relationship operator is used the result set may contain some unnecessary results which they call **false drops**. Therefore, a refinement step is also needed.

Through their performance evaluation, the authors find that semijoin algorithms reduced the cost of evaluating a join in most cases. For the R-tree based approximation of objects, they found that for the Geographic information Systems (GIS) applications, a large number of approximations is preferred while for Land Information System (LIS) applications, a smaller number of approximation provides better performance. They also found that, index nested loop join is preferred over R-Tree based indexing if the index needs to be constructed as R-tree construction is expensive. In addition, locational key based algorithms, do not lead to significant performance gains, but do perform better on LIS applications, over GIS applications.

The authors identify the following research directions. 1) Minimize the high CPU cost from R Tree based joining algorithms, 2) Evaluating buffering techniques, and 3) Examining the performance of locational key based algorithms by applying approximation with them.

Optimizing Distributed Spatial Semijoin

Karam and Petry [13] proposed a distributed spatial semijoin operator that takes MBRs from different levels of the R-tree, instead of from the same level or the leaf level of the tree. A performance comparison versus the traditional distributed spatial join shows that with respect to data transmission cost, their semijoin is superior when applied to real world data.

Distributed Spatial Query Optimization Over Multiple-Sites

Osborn and Zaamout [14] proposed a general distributed query processing strategy that utilizes MBR-based distributed spatial semijoins, and worked for queries that involved more than two sites. The strategy transmits the smaller spatial attributes to the sites that contain larger relations. After the semijoin is performed on those sites, only the identifiers are then transmitted back to the originating (smaller) sites, and all qualifying tuples are sent to the query site for the final spatial join. An evaluation of their strategy with two-, four-, and six-site queries on a simulated distributed database, found that the strategy achieves lower data transmission costs.

In addition, Osborn and Zaamout [25] proposed a strategy for optimizing distributed spatial semijoin which they call “restricted strategy”. Instead of sending all the *MBR*s, they sent only the group approximations of them, which is an improvement upon the strategy mentioned in [14], as it reduces CPU and transmission costs.

2.4.5 Summary

All the approaches dicussed above have one or more of the following limitations: 1) the algorithms were designed for only two sites, 2) the algorithms which were designed for more than two sites are only simulated on one machine or on local cluster environment, 3) there were no attempts made to compare the algorithms with the real world Distributed Naïve Spatial Semijoin, 4) there was lack of consideration of all of the vital cost factors (i.e. CPU, I/O and transmission cost).

Chapter 3

GEMBR Partitioning and Mapping

This chapter presents the core algorithms, which are used by the compact representations of the distributed semijoins. The first is the calculation of a Global Encompassing Minimum Bounded Rectangle (GEMBR) using the *GEMBR Calculation Algorithm*. After calculation, the GEMBR is partitioned and all the object MBRs are mapped onto it using the *GEMBR Partitioning and Mapping Algorithm*. The GEMBR will contain all the objects MBRs from all the relations of all sites.

3.1 GEMBR Calculation

The Global Encompassing MBR (GEMBR) is the spatial extent of all objects that exist across all sites in the distributed spatial database. Figure 3.1 shows the overall sequence of steps for calculating the GEMBR. The steps for the GEMBR calculation algorithm are as follows:

1. The lower left and upper right coordinates (i.e. (lx, ly) and (hx, hy) respectively) of the Local Encompassing MBRs (LEMBR) are obtained at each site. A Local Encompassing MBR (LEMBR) is the spatial extent of all spatial objects (or their MBRs) that exist at one site. If an R-tree exists at a site, this LEMBR can be extracted from the root node of the tree. This is shown in Figure 3.2.

2. Next, the GEMBR is calculated as follows. The LEMBRs from each of the sites are sent to a query site. All the (lx, ly, hx, hy) values of all the LEMBRs are divided into four sets: lx, ly, hx, and hy and each set is sorted. From these ordered sets, the lowest (lx, ly) and the highest (hx, hy) coordinates are identified. These are the resulting

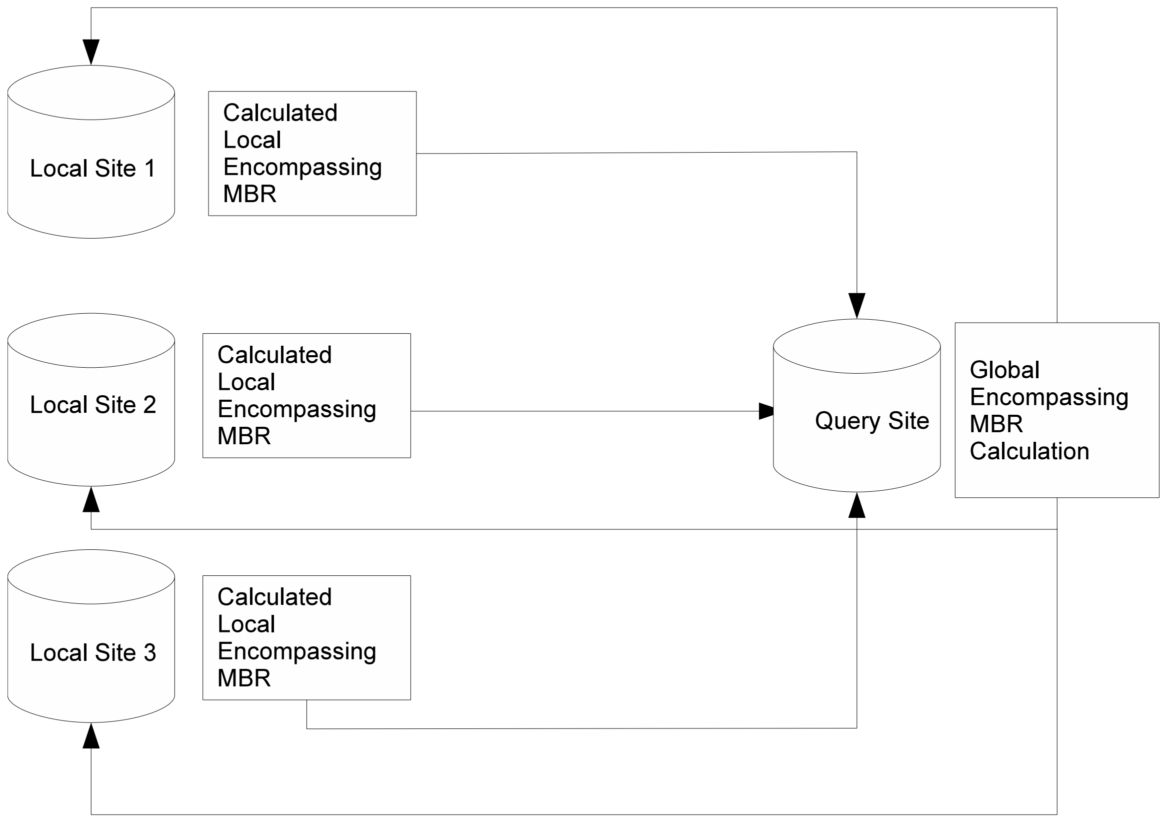

Local Site 1
Calculated Local Encompassing MBR
Local Site 2
Calculated Local Encompassing MBR
Query Site
Global Encompassing MBR Calculation
Local Site 3
Calculated Local Encompassing MBR

Figure 3.1: GEMBR Partitioning at Client Sites

GEMBR coordinates, which are sent to each of the sites in parallel This is shown in Figure 3.1

For example, suppose we have four LEMBRs from four sites having the following (lx, ly, hx, hy) coordinates: $(0,0,4,2)$, $(1,1,6,3)$, $(2,2,8,4)$ and $(3,3,10,5)$. Now, if we sort all the values in the lx set in ascending order we find following values: 0, 1, 2 and 3. We take the first value, (i.e. 0) as the minimum lx value among all values in the lx set. Likewise, we also find ly, hx and hy values as 0, 10 and 5 respectively. So, the final GEMBR coordinate (Lx, Ly, Hx, Hy) is $(0,0,10,5)$. This example is shown in Figure 3.3.

3. Then, the GEMBR is partitioned at each of the sites. Using the partition information

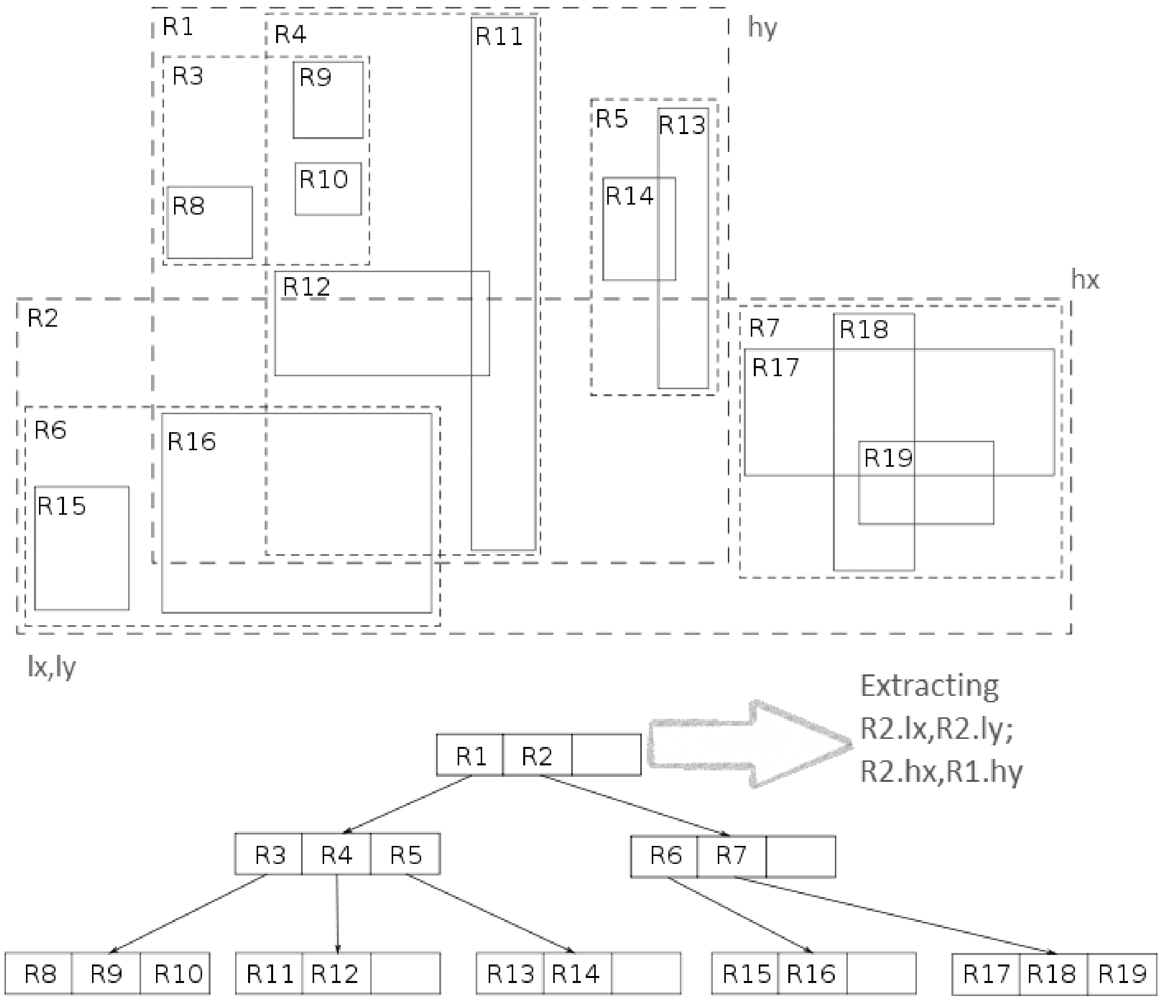


Figure 3.2: Extracting LEMBR Coordinates

n sent by the query site, the copy of the GEMBR at each client site is partitioned into $n \times n$ partitions, and indexed from lower left corner to upper right corner using positive integer i, where $0 \leq i \leq n \times n$, which we call partition indices. Currently, the partition information n is a constant value, which is stored at the query site, or can be provided by the user when specifying a query. This is shown in Figure 3.4.

4. Finally the object MBRs at each client site are then mapped onto the partitioned GEMBR. If an R-tree is used, the object MBRs can be obtained from its leaf nodes, which results in lower I/O costs. This is shown in Figure 3.5

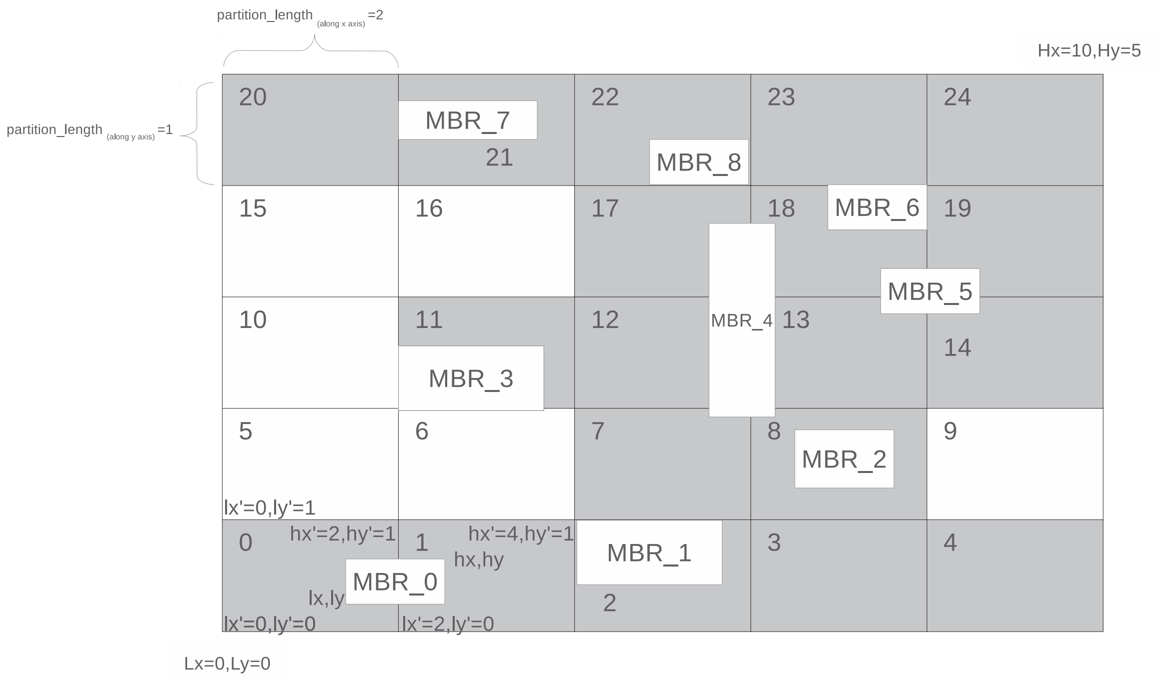


Figure 3.3: GEMBR Calculation, Partitioning and Mapping Example

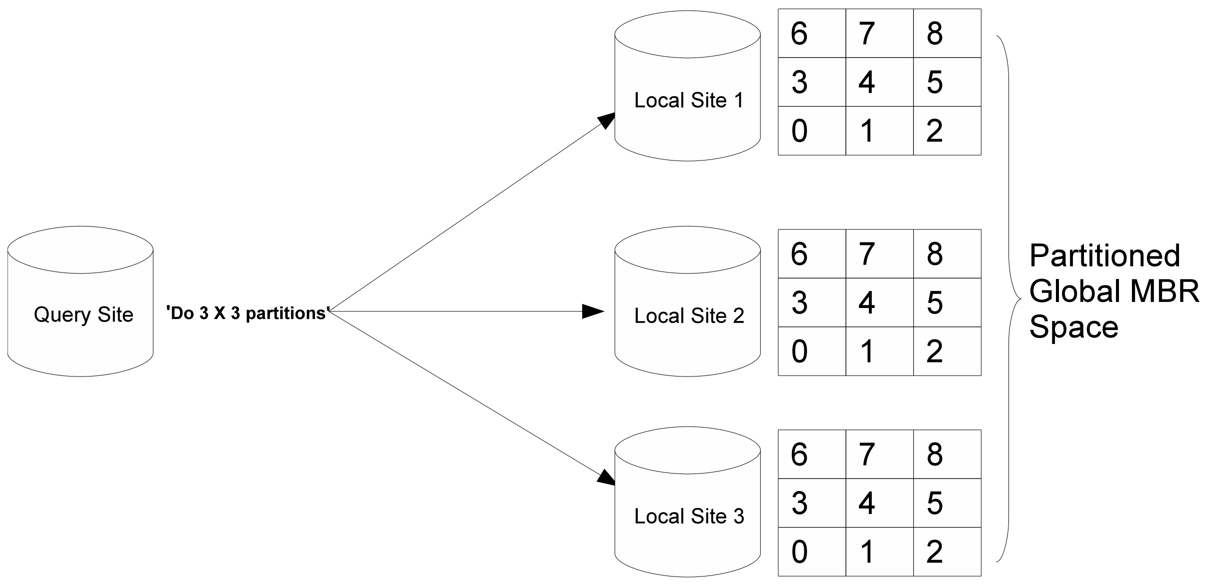


Figure 3.4: Partitioned GEMBRs at Each Site

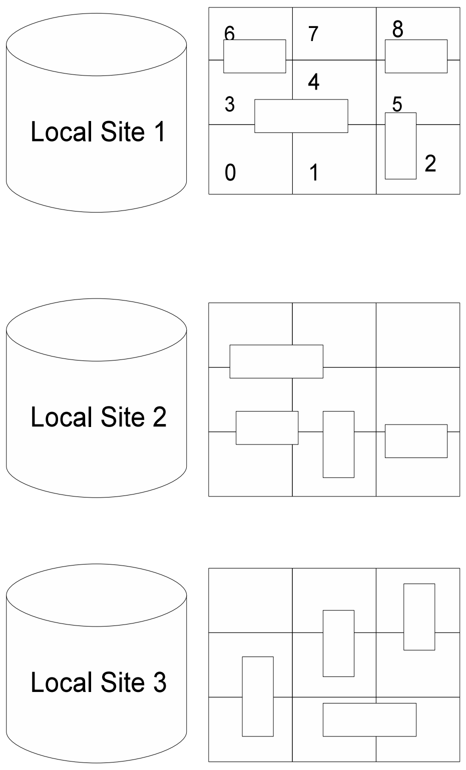


Figure 3.5: Mapping Objects MBRs on Partitioned GEMBR at Each Site

The partition and mapping algorithm, which is briefly described in Steps 3 and 4 above is described in more detail in the next section.

3.2 GEMBR Partitioning and Mapping

After the GEMBR is calculated and transmitted to each site, the following GEMBR partitioning and mapping algorithm is carried out on each site in parallel to partition the GEMBR space and map the local object MBRs to the local GEMBR copy. The steps for this are as follows:

1. First, the GEMBR space is partitioned into n partitions along the x axis and n partitions along the y axis. Therefore, the total number of GEMBR partitions, $\#partitions$, is $n \times n$. The length of each partition along the x axis and y axis is calculated.

```
Algorithm: Create_Partitions

Input:
    GEMBR Lx, Ly, Hx, Hy
    num_partitions_along_each_axis n

Output:
     holder_array[nxn]

partition_index= 0

ly = Ly

while ly  <= Hy

    lx = Lx

    while lx <= Hx

        lx’ = lx
        ly’ = ly
        hx’ = lx + partition_length(n, abs(Hx - Lx))
        hy’ = ly + partition_length(n, abs(Hy - Ly))

        holder_array[partition_index]
             = (lx’, ly’, hx’, hy’)

        partition_index ++

        lx += partition_length(n, abs(Hx - Lx))

    end while

    ly += partition_length(n, abs(Hy - Ly))

end while

return holder_array
```

Figure 3.6: Creation of Partitions

Referring to the GEMBR example shown in Figure 3.3, where the GEMBR has the lower coordinate $(Lx = 0, Ly = 0)$ and upper coordinate $(Hx = 10, Hy = 5)$. Also, assume that the number of specified partitions n along an axis is 5. We need to create $5 \times 5 = 25$ partitions. Therefore, the length of each partition along x axis is 2, and along the y axis is 1.

2. Next, the coordinates $(lx\prime, ly\prime, hx\prime, hy\prime)$ for each partition is calculated using the algorithm in Figure 3.6. The process proceeds through the partitions in row-major order, starting from the lower left-hand corner (Lx, Ly) to the top right-hand corner (Hx, Hy) of the GEMBR. After each partition is calculated, it is stored in an array, which is called *partition coordinates holder array* (or *holder array* for short). The index values for the *holder array* will serve as the partition identifiers later on.

   Referring back to Figure 3.3, the lower left partition is calculated first (i.e $lx\prime = 0$, $ly\prime = 0$, $hx\prime = 2$, $hy\prime = 1$), followed by the next partition (i.e. $lx\prime = 2$, $ly\prime = 0$, $hx\prime = 4$, $hy\prime = 1$), and proceeding towards the upper right-hand partition, in row-major order.

3. Next, the object MBRs are mapped onto the local copy of the GEMBR. First, for each object MBR, the GEMBR partition (or subset of partitions, which we will call the GEMBR subregion) that encompass the object are calculated. Because an object can overlap more than one partition, this step determines the entire region covered by the subset of partitions that contain an object.

   For each object MBR, its lower left coordinate (lx, ly) and upper right coordinate (hx, hy), are tested against the lower left-hand and upper right-hand coordinates of each of the partitions, starting from the lower left-hand partition 0 and proceeding in row-major order to up the upper right-hand parition. If the lower coordinate (lx, ly) is inside any partition, or on either of the partition coordinates ($lx\prime$,$ly\prime$) or ($hx\prime$,$hy\prime$), the lower left coordinates of a partition $(lx\prime, ly\prime)$ is recorded as the lower left-hand coordinate of the GEMBR subregion that encloses the object MBR. Similarly, the upper right coordinates ($hx\prime$,$hy\prime$) of a partition are recorded as the upper right-hand coordinate of the GEMBR subregion if it contains the upper right (hx, hy) coordinate of an object MBR.

Referring back to the example in Figure 3.3, suppose we test the lower left coordinates (lx, ly) of MBR_0 and find they are inside the lower left $(lx\prime = 0, ly\prime = 0)$ and upper right $(hx\prime = 1, hy\prime = 2)$ coordinates of partition 0 (i.e. its $(lx\prime, ly\prime) \leq (lx, ly) \leq (hx\prime, hy\prime)$). We record ($lx\prime$ = 0,$ly\prime$ = 0) from partition 0 as the lower left-hand coordinate of the GEMBR sub region containing MBR_0. Then we check the upper right coordinate (hx, hy) of MBR_0 and find that it falls inside partition 1. We record as the upper right-hand coordinate of the GEMBR subregion the upper right hand coordinate($hx\prime = 4$ and $hy\prime = 1$) from partition 1. Therefore, these lower left-hand and upper right-hand coordinates define the subregion of partitions that MBR_0 encompasses. For MBR_4, its lower left corner lies in between the lower left and upper right corners of partition 7 and similarly its upper left corner is situated inside partition 18. Hence, all the partitions within the range of the lower left corner of partition 7 and upper right corner of partition 18 defines the GEMBR subregion for MBR_4.

4. For the identification of partitions overlapped by an object, we take each object MBR and traverse through the *holder array* in row major order and determine if any of the corner points of that MBR space(i.e. either the lower left or upper right) fall in between any of the lower and upper coordinates for a partition. If the lower left coordinates of an MBR falls inside a the lower left and upper right coordinates of a partition, we record lower left coordinates for that partition in an array named *gembr subregion array*. The same test is performed for the upper right coordinates of an object MBR and the upper right coordinate component of *gembr subregion array*

is updated. Therefore, all the partitions which have their lower left and upper right corners in between the lower and upper coordinates of the *gembr subregion array* for the object MBR make up the GEMBR subregion and the partition indices of those partitions are returned. Referring back to Figure 3.3, *MBR_*0 is mapped into partitions 0 and 1. *MBR_*2 is mapped into partition 8, *MBR_*4 to partitions 7, 8, 12, 13, 17 and 18, *MBR_*5 to partitions 13, 14, 18 and 19, *MBR_*6 to partitions 18, 19, 23 and 24, *MBR_*7 to partitions 20 and 21 and *MBR_*8 to partitions 22 and 23. Finally the set of unique partition indices returned by all the object MBRs are saved for final mapping.

3.2.1 Corner Cases

The four corner cases (i.e. where an object has its lower left or upper right coordinates exactly on the any two corner points of a partition) is discussed in more detail here. Corner cases are critical as each has a varied number of qualified partitions which changes depending on the case. For all the examples discussed below please refer to Figure 3.3. For example, *MBR_*6 is located in the upper right corner of partition 18. According to the algorithm, the *gembr subregion array* is first updated by the lower left coordinates of partition 18 as the lower left coordinates of *MBR_*6 resides between the lower left and upper right coordinates of partition 18. Through out the traversal, the lower left coordinates are not updated as they are not inside any of the partitions from 19 to 24. On the other hand the upper right corner of *MBR_*6 resides in partitions 18, 19, 23 and 24. When the algorithm passes through partition 19, the array is updated for the second time with the upper right coordinates. This array is updated two more times with the upper right coordinates of partition 23 and 24 respectively. Finally, we have the final array which consists of the lower left and upper right coordinates of partition 18 and partition 24 respectively. Therefore, for

*MBR*_6 the partitions 18,19,23 and 24 are returned.

For *MBR*_8, which is at the lower left corner of partition 22, identifies partitions 22 and 23. Initially, its lower left coordinates falls inside partition 17, and its upper left coordinate falls inside partition 23. But with further traversal, the coordinates are updated, and finally the *gembr subregion array* holds the lower left and upper right coordinates of partition 22 and 23. Similarly *MBR*_3 falls inside the lower left corner of partition 11, and overlaps partition 11. Our last such case *MBR*_1 overlaps partitions 2 and 7 following the same procedure.

Chapter 4
Query Processing Strategies

In the following section, three distributed query processing algorithms are proposed, two of which utilize our compact representations of the distributed spatial semijoin:

1. Geometric Space Partition and Mapping Based Spatial Semijoin (PMSJ),
2. Bloom Filter Based Spatial Semijoin (BFSJ), and
3. Distributed Naïve Spatial Semijoin (NSPJ).

The algorithms are designed to work for any number of sites.

Among the sites, one is designated as the query site where the user issues a query. All other sites are the client sites which process a portion of the user query using data that is stored locally. All the processes initiate at each site at the same time when the user issues query from query site. In the user query, the user states the number of partitions *n* along an axis of the GEMBR for both algorithms. For the BFSJ algorithm, the *bloom filter factor* and *number of hash functions* are also stated.

4.1 Geometric Space Partition and Mapping Based Spatial Semijoin (PMSJ)

The first algorithm, PMSJ, represents a semijoin by utilizing the partition indices representation of the GEMBR from all participating sites. PMSJ comprised of the following steps:

1. On each client site, the partition indices are obtained from the GEMBR Calculation,

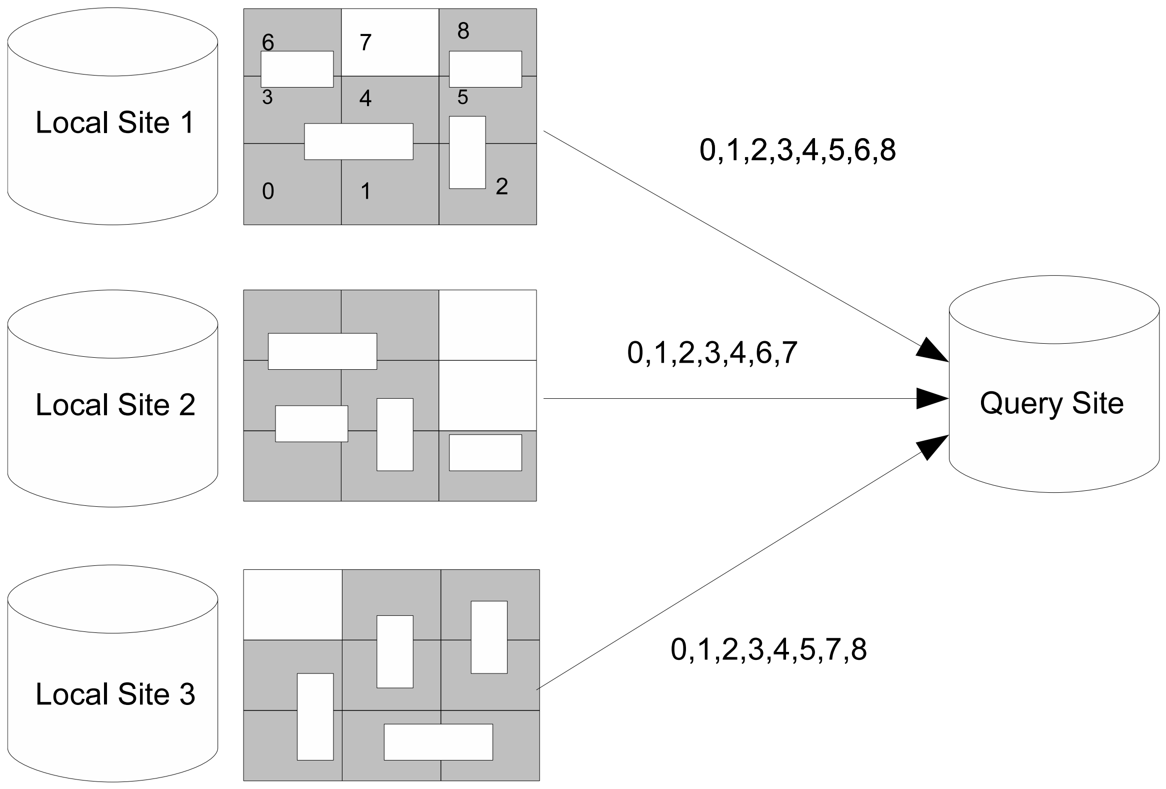


Figure 4.1: Sending Partition Indices to Query Site

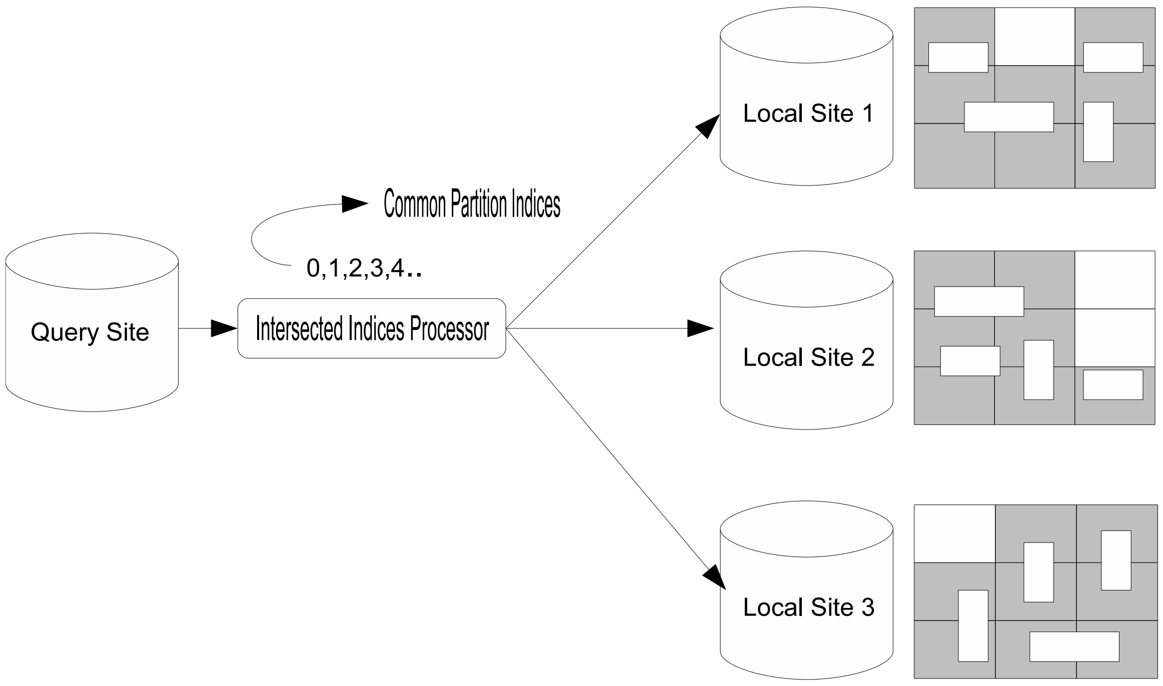


Figure 4.2: Calculating and Transmission of Common Partition Indices

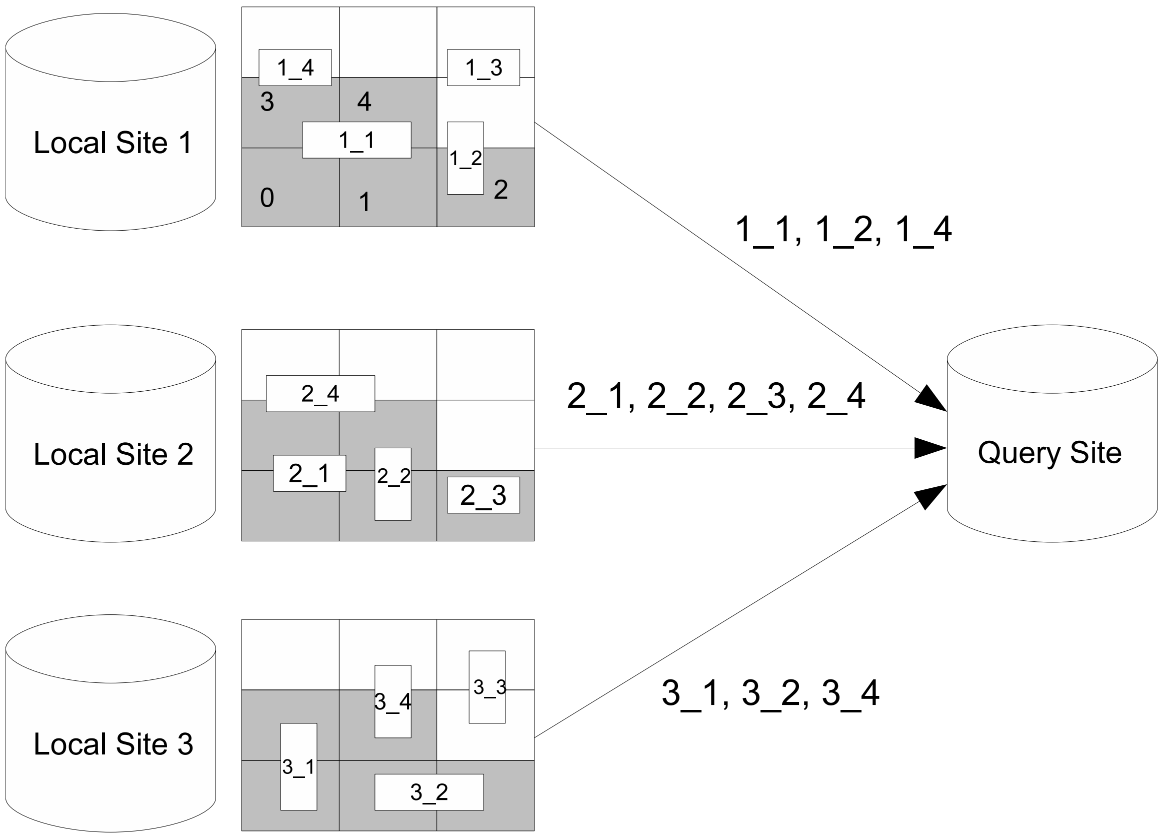


Figure 4.3: Object Transmission to Query Site

Partition and Mapping algorithms, and duplicates are removed before transmission to the query site. This is shown in Figure 4.1.

2. At the query site, the set of common partition indices is calculated. A partition index is added to this set only if it was sent from every client site (i.e. at every client site, the partition contained one or more objects). Then, the final set of common indices are sent back to each client site. This is shown in Figure 4.2.

3. On each client site, all the tuple ids of the corresponding object MBRs that reside in the partitions contained in the set of common partition indices are retrieved. Finally, on each client site, for each qualifying tuple id, the corresponding exact spatial object is retrieved and sent to the query site for the refinement step. This is depicted in Figure 4.3. Note that in Figure 4.3 each spatial object is represented with its tuple id due to limited space in the diagram - however, it is the objects that are being transmitted to the query site.

4.2 Bloom Filter Based Spatial Semijoin (BFSJ)

The Bloom Filter Based Spatial Semijoin (BFSJ) creates and uses Bloom filter based representations of the GEMBRs from all participating sites for semijoin processing. The steps for BFSJ are as follows:

1. According to the query information, which consist of the number of the partitions along each axis n, the number of hash functions k and the Bloom filter factor B_f specified by the user from the query site, the size of the Bloom filter is calculated at each of the client sites based on following formula: $B_s = B_f \times (n \times n)$. Then, all the Bloom filters are initialized to all 0s.

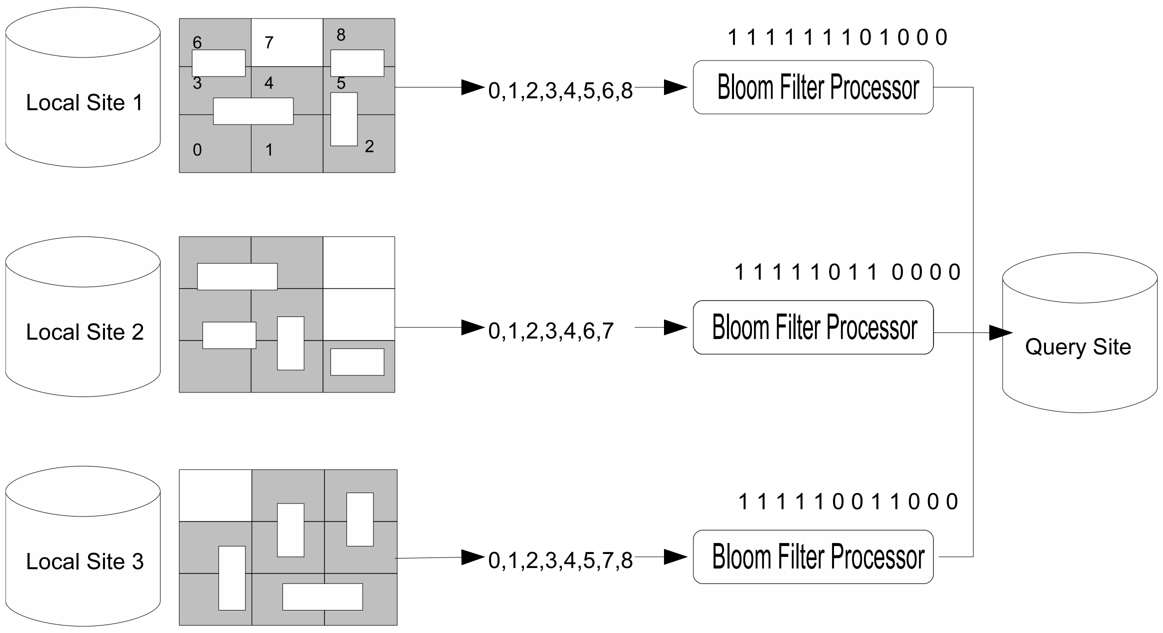


Figure 4.4: Mapping Partition Indices to Bloom Filter

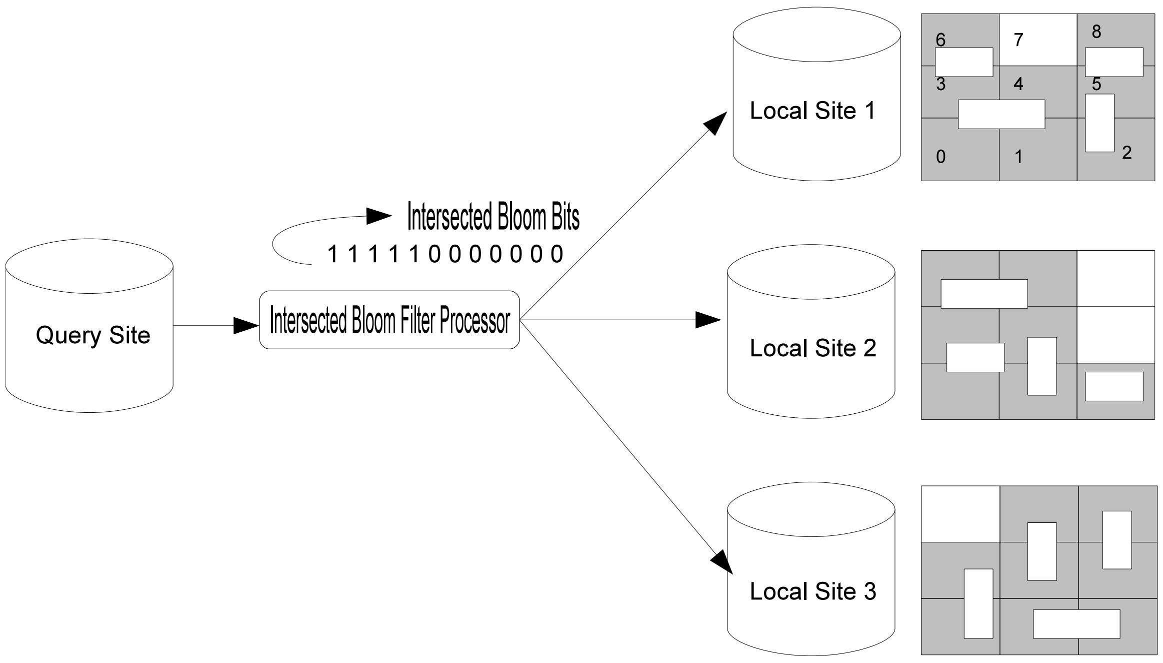


Figure 4.5: Calculating and Transmission of Common Bloom filter

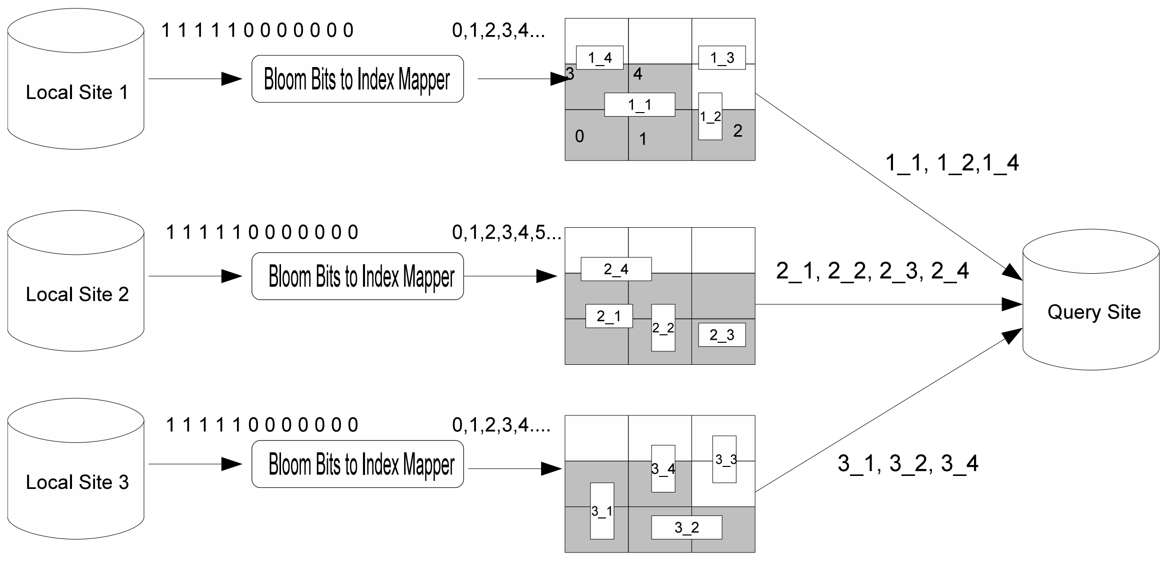


Figure 4.6: Bloom Bit to Partition Index Mapping and Object Transmission to Query Site

2. The Bloom filter representations of the partition indices from each site are constructed and sent to the query site. Bloom filter creation is done by a module called the *Bloom Filter Processor*. For this, the *Bloom Filter Processor* takes the set of partition indices that are returned from the GEMBR Calculation, Partitioning and Mapping functions, and sends it to the hash functions. The hash functions calculate and return the corresponding Bloom filter indices, which are then set to 1. This is shown in Figure 4.4.

3. At the query site, the *Intersected Bloom Filter Processor* module finds the common Bloom filter by performing a bit-wise intersection of all Bloom filters in order to find out the common Bloom bits from all the Bloom bits representations and send the common or intersected bloom bits to each of the client sites. This is depicted in Figure 4.5.

4. At each of the client sites the common Bloom filter bits are sent to the *Bloom Bits to Index Mapper* module, which maps the Bloom filter bits to the corresponding partition indices. This is done by taking each of the partition indices and if it is found to be 1 in the Bloom filter (after being hashed by all the hash functions), it is kept. These partition indices become the common partition indices. This is shown in Figure 4.6.

5. Next, at each client site, all the tuple ids of the corresponding object MBRs that reside in the partitions contained in the set of common partition indices are retrieved. Finally, on each client site, for each qualifying tuple id, the corresponding exact spatial object are retrieved and sent to the query site for the refinement step. This is also depicted in Figure 4.6. Note that in Figure 4.6 each spatial object is represented with its tuple id due to limited space in the diagram - however, it is the objects that are being transmitted to the query site.

4.3 Distributed Naïve Spatial Semijoin (NSPJ)

The Distributed Naïve Spatial Semijoin (NSPJ) algorithm is an extension of [22] for more than two sites. We use this as a benchmark strategy for comparison purposes. The steps for this algorithm are as follows:

1. All the object MBRs at each client site are sent to the query site. If the spatial relation at a client site is indexed by an R-tree, then the object MBRs can be obtained by scanning the leaf nodes of the R-tree.

2. At the query site, all object MBRs from all sites are checked for overlap. This is done by testing every set of object MBRs across all spatial attributes that are transmitted to the query site.

3. The qualifying object MBRs from step 2 are sent to all the client sites. Their corresponding tuple ids are extracted. Then, all the qualifying exact spatial objects for those tuple ids are sent to the query site for the refinement step.

Chapter 5

Experimental Evaluations

In this chapter the performance evaluation of the BFSJ, PMSJ and NSPJ strategies is presented. The first goal is to compare the BFSJ and PMSJ strategies against the NSPJ strategy with respect to processing and data transmission costs. The second goal is to compare the BFSJ and PMSJ strategies with respect to false drops. The third goal is to compare different BFSJ algorithms having different number of hash functions and Bloom filter factors. In addition, all four strategies are evaluated in a real life multi-site environment where other proposed approaches only consider two sites or a simulated multi-site environment when processing a distributed query.

5.1 Preliminaries

The algorithms are implemented in a four node peer-to-peer distributed system[1]. The nodes are situated in four geographically scattered locations. Our query site is located at the University of British Columbia, and the client sites are located at the University of Victoria, Simon Fraser University and the University of Alberta.

The evaluation was carried out with both synthetic data that was generated randomly[2], and with real data that was obtained from the Sequoia 2000 benchmark [21]. Synthetic data was utilized so that results and trends could be determined for specified numbers of tuples and a random data distribution. In addition, it is important to show how the algorithms perform with real-world data.

Each relation is indexed on its spatial attribute with an R-tree[3]. In addition, a parallel

[1]Westgrid, www.westgrid.ca
[2]Generator created by Marc Moreau, The University of Lethbridge
[3]R-tree code obtained from www.rtreeportal.org.

distributed shell (PDSH) utility[4] was used for co-ordinating the overall spatial semijoin process in parallel at each client site, as soon as the user issues the query from the query site. All algorithms are implemented in C++, and total process to be carried out in the distributed environment is organized with the help of Bash shell scripting.

The BFSJ, PMSJ and NSPJ algorithms are compared with respect to the processing (CPU+I/O) time and data transmission cost. The CPU+IO time is measured in seconds, while the data transmission cost is measured in the number of KiloBytes that are transmitted over the network. BFSJ and PMSJ algorithms are compared based on the percentage of false positives. As the NSPJ uses the object MBRs directly for overlap checking, it does not produce any false positive result in the filter stage. Therefore, the percentage of the false positives is calculated only for BFSJ and PMSJ. The percentage of false positives is calculated based on the following formula:

$$f = \frac{\#Tuples_{BFSJorPMSJ} - \#Tuples_{NSPJ}}{\#Tuples_{BFSJorPMSJ}} \times 100$$

In addition, the BFSJ algorithm is evaluated using up to 10 hashfunctions with different Bloom filter factors.

5.2 Results using Synthetic Data

For this set of tests, each client site contains five spatial relations that contain 2000, 4000, 6000, 8000 and 10,000 tuples respectively. Every spatial relation contains one spatial attribute, which consists of 10-unit by 10-unit squares. For each relation, a space size of $\sqrt{\#tuples} * 10$ units is used to contain all randomly generated squares.

The comparison of processing time (PT items in each legend) and average transmission

[4]PDSH utility obtained from code.google.com/p/pdsh

cost (TC items in each legend) is shown for 30x30, 60x60 and 90x90 partitions in Figures 5.1, 5.2, and 5.3 respectively. The false positive percentage comparison is shown in Figures 5.4, 5.5 and 5.6 for the same partitions. From the figures, we observe a common trend of increasing transmission cost, and increasing processing time, and decreasing false positive percentages with the increase in the number of partitions.

5.2.1 Transmission Cost

The average transmission cost in KiloBytes(KB) (the "inefficiency index" measure for transmission cost, as displayed in the Figures) of data transmitted from client sites to query site and vice-versa is calculated in parallel. This data is plotted against the number of tuples in the figures. By comparing the three algorithms, it is found that the PMSJ and BFSJ algorithm outperforms NSPJ algorithm by a factor of approximately six on average with respect to transmission cost. For example, in Figure 5.3, a significant difference is found in transmission costs between both the BFSJ and PMSJ algorithms and the NSPJ algorithm. This cost for NSPJ is very high with respect to number of tuples when compared to BFSJ and PMSJ. This observation is true as well for Figures 5.1 and 5.2. The reason is that more compact approximations of MBRs are sent to the query site from the local sites in the algorithms, where in NSPJ the actual MBR approximations of the spatial objects are sent. The transmission cost is further reduced by only sending unique partition indices which are covered by object MBRs on a client site to the query site for semijoin processing or to the *Bloom filter processor* for Bloom filter creation.

We also compare both the BFSJ and PMSJ algorithms and observe that these algorithms perform very closely, with BFSJ having a transmission time gain of approximately 1.12 over PMSJ. This is due to sending Bloom filters, which only contain boolean values and are more compact than the partition indices. This is seen in Figure 5.1 for 30×30 partitions

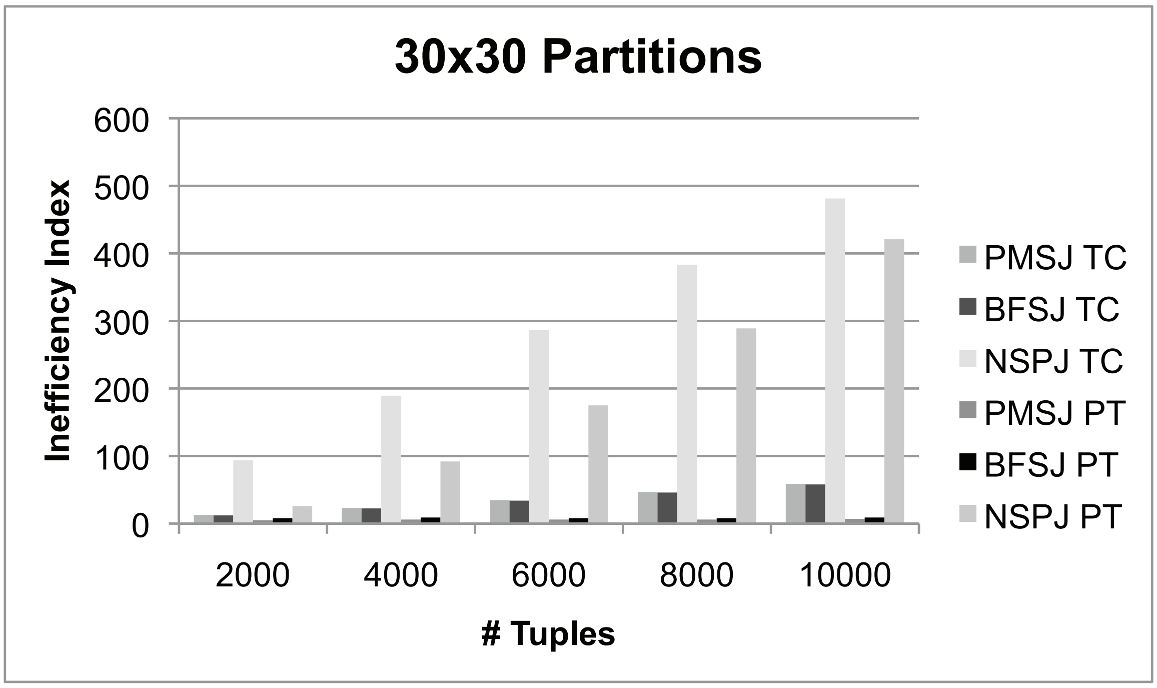


Figure 5.1: Processing Time and Transmission Cost for 30 × 30 Partitions

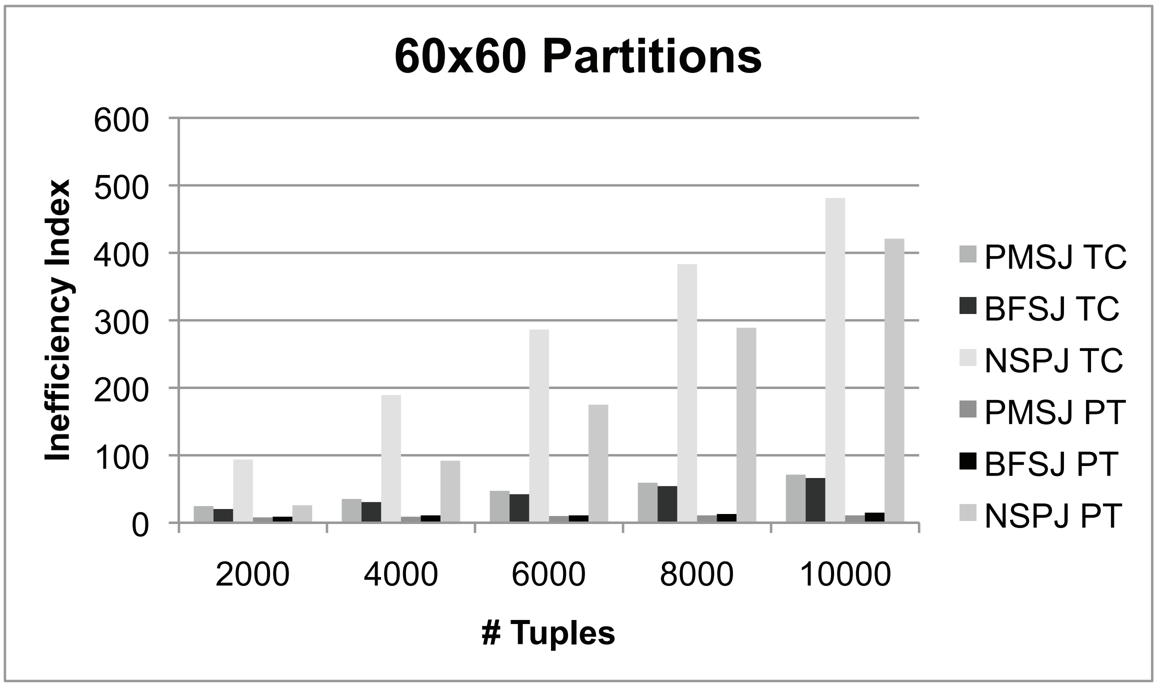


Figure 5.2: Processing Time and Transmission Cost for 60 × 60 Partitions

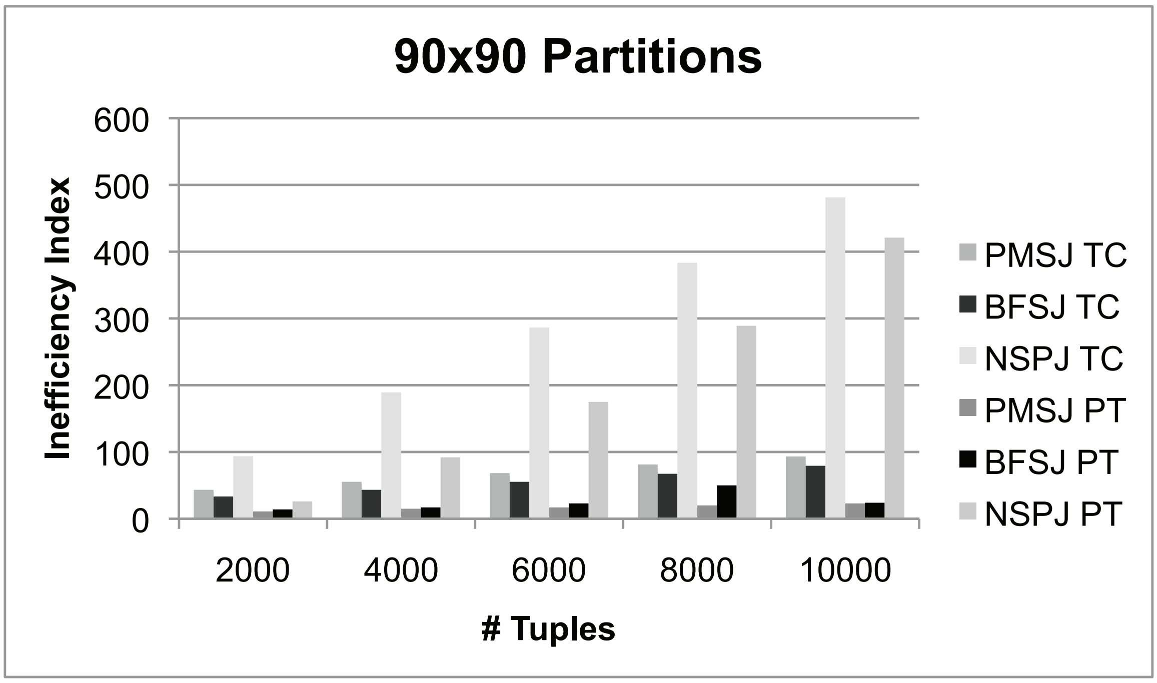


Figure 5.3: Processing Time and Transmission Cost for 90 × 90 Partitions

where both strategies are equal, and in Figures 5.2 and 5.3 respectively where we can see a little performance gain of BFSJ over PMSJ for 60 × 60 and 90 × 90 partitions. The overall transmission cost increases linearly with the increase of the number of partitions (e.g. transmission cost increases roughly 1.9 times when partition number increases three times) and number of tuples (e.g. transmission cost increases 3.3 times when number of tuples increases five times) in BFSJ and PMSJ. This happens due to the fact that more partitions results in more data transmission. This is observed in Figures 5.1, 5.2 and 5.3.

5.2.2 Processing Time

For processing time, the amount of time the algorithms execute in seconds (the "inefficiency index" measure for processing cost, as displayed in the Figures), which includes both the CPU and the I/O time. This is plotted against the number of tuples. Both BFSJ and PMSJ are on average 18 times faster than NSPJ for 30×30, 60×60 and 90×90 partitions in Figures 5.1, 5.2 and 5.3 respectively. For example, Figure 5.1 shows that there is a significant difference between BFSJ, PMSJ and NSPJ. This is because in the BFSJ and PMSJ algorithms, more compact representations are utilized for semijoin processing, which are only integers or boolean values, while in NSPJ the actual object MBRs are utilized, which ultimately incurs extra CPU time.

The performance gain of PMSJ over BFSJ, is approximately on average 1.38. This is because there is extra processing for creating and setting bits in the Bloom filters, and remapping partition indices from the Bloom filters. This observation is consistent for both 60×60 and 90×90 partitions. The increase of the processing time is linear with respect to the increase in number of partitions, as the semijoin operates on more data. For example, if we increase the number of partition three times, the increase of processing time is 2.9 times. This is also observed in the Figures 5.1, 5.2 and 5.3.

5.2.3 False Positive Comparison

With respect to the false positive percentages, the difference between BFSJ and PMSJ is more prominent with the increase in number of partitions. For example, there is a tie between PMSJ and BFSJ in case of 30×30 partitions, which is depicted in Figure 5.4. However, the superiority of PMSJ becomes significant when the number of partitions increases as depicted in Figure 5.5 and Figure 5.6. PMSJ always outperforms BFSJ by a factor of 1.02 on average. With the increased number of partitions, the increase in the false positive percentage is linear with a gain of 1.02 (which means the false positive percentage decreases) on average for both of the PBSJ and BFSJ algorithms, which is also observed in all the three Figures. With more partitions we obtain a more accurate mapping of object MBRs in the GEMBR and thus the false positive percentage decreases.

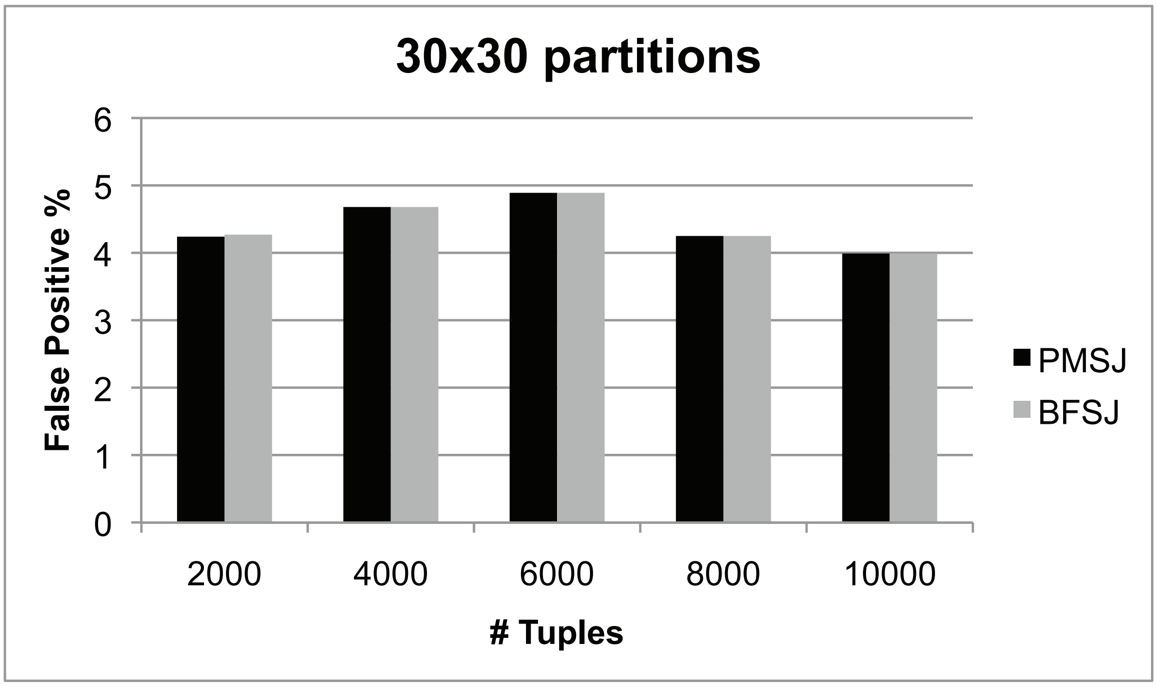


Figure 5.4: False Positives Comparison for 30 × 30 Partitions

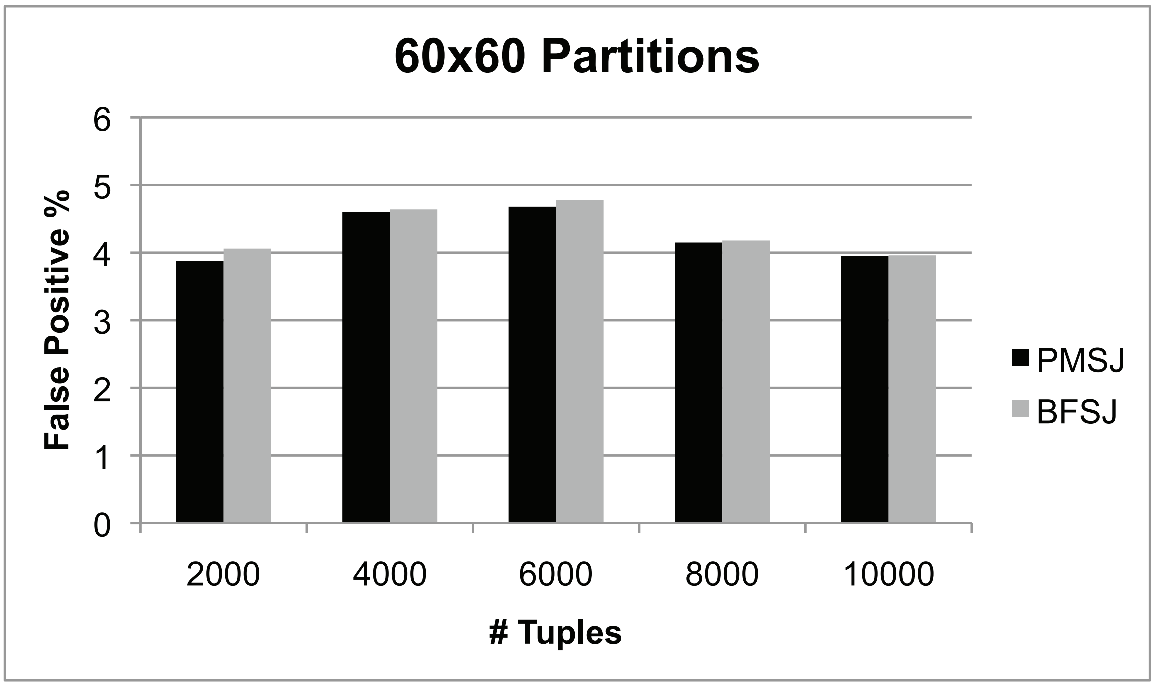


Figure 5.5: False Positives Comparison for 60 × 60 Partitions

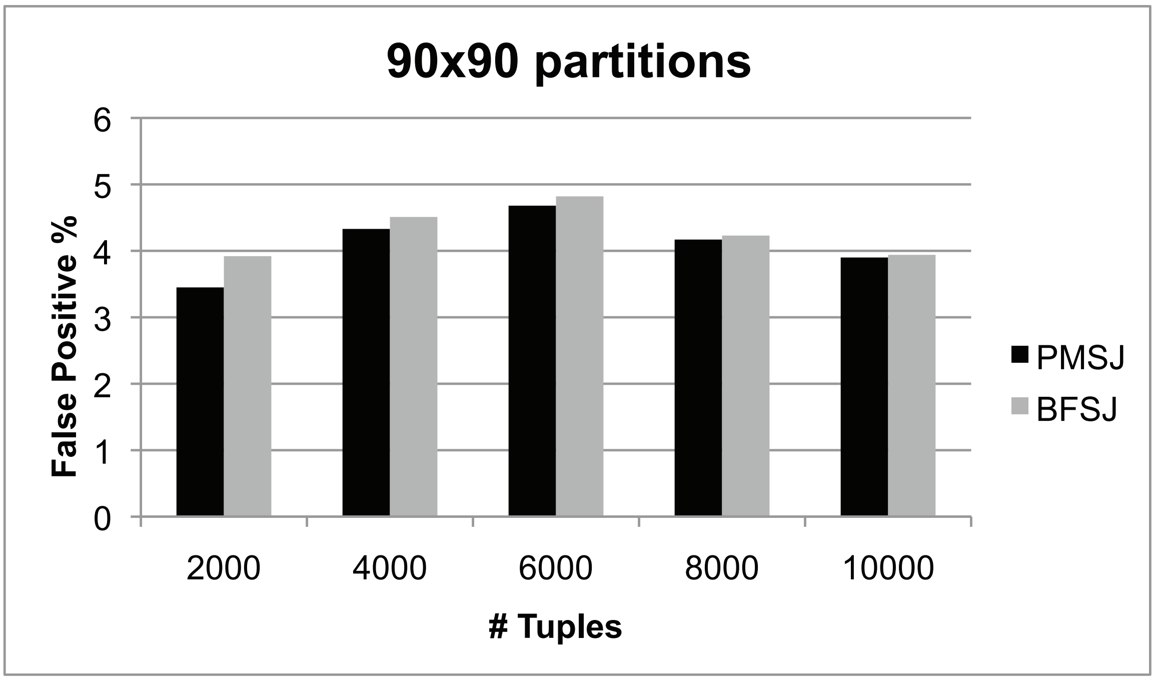


Figure 5.6: False Positive Comparison for 90 × 90 Partitions

PMSJ predominates in this case because, BFSJ takes the partition indices as its input so it has at least the false positive percentage as the PMSJ along with the false positive hits of its own.

5.3 Results using Sequoia Data

For these tests, MBRs are created from the polygons of the Sequoia 2000 benchmark data set [21]. The polygon set consists of landuse polygons (58586 MBRs) and islands, which represent "holes" in the landuse polygons (21021 MBRs). Two tests are performed with these data sets. For the first test, the algorithm is evaluated by placing the islands dataset at one site, and spliting the landuse dataset into two sets (48778 MBRs and 9808 MBRs) to place at the remaining two sites. Two tests are run, using 90x90 and 300x300 partitions respectively. For the second test, the landuse polygon set is split into three sets

#partitions	strategy	Processing Time (s)	Transmission Cost (kb)	FP Rate (%)
90 x 90	PMSJ	15	264	79.51
	BFSJ	17	275	80.31
	NSPJ	2040	2865	——
300 x 300	PMSJ	128	149	63.57
	BFSJ	139	178	69.64
	NSPJ	2040	2865	——

Table 5.1: Sequoia Results - Polygon Datasets

#partitions	strategy	Processing Time (s)	Transmission Cost (kb)	FP Rate (%)
90 x 90	PMSJ	33	355	84.04
	BFSJ	40	373	84.80
	NSPJ	1009	3879	——
300 x 300	PMSJ	216	186	69.46
	BFSJ	226	232	75.60
	NSPJ	1009	3879	——

Table 5.2: Sequoia Results - Polygon and Islands Datasets

- 28728, 9808 and 20050 MBRs respectively - one per site. Again, we executed each algorithm twice - for 90x90 and 300x300 partitions. Tables 5.1 and 5.2 gives the results of this evaluation. We find that both the PMSJ and BFSJ algorithms achieve significantly lower transmission costs than the NSPJ algorithm. However, the percentage of false positives is very high for both the PMSJ and BFSJ algorithms. This is due to MBRs polygons across multiple sites that are close in proximity but do not overlap. Due to the close proximity, the MBRs would appear in the same partition and be transmitted to the query site. Therefore, our algorithms, although shown to perform very well on our synthetic data sets, would benefit further by adapting to the distribution of the object sets across all sites.

5.4 Number of Hash Functions vs Percentage of False Positives

The BFSJ algorithm was executed for up to 10 hash functions with Bloom filter factors 1.5 and 3 to observe how false positive percentage changes. A common trend of increasing false positive percentage is observed with the increased number of hash functions. However, the false positive percentage begin to plateau when 7 or more hash functions are used. This is depicted in Figure 5.7 and 5.8.

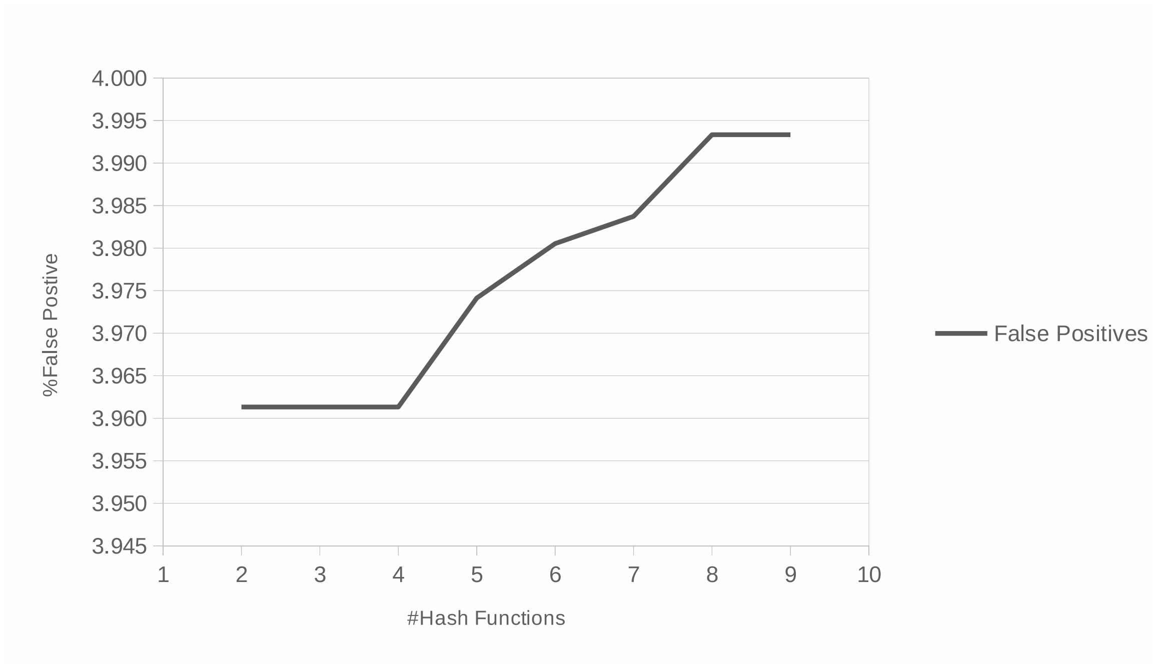


Figure 5.7: False Positive Comparison for Different Number of Hash Functions with Bloom filer Factor 1.5

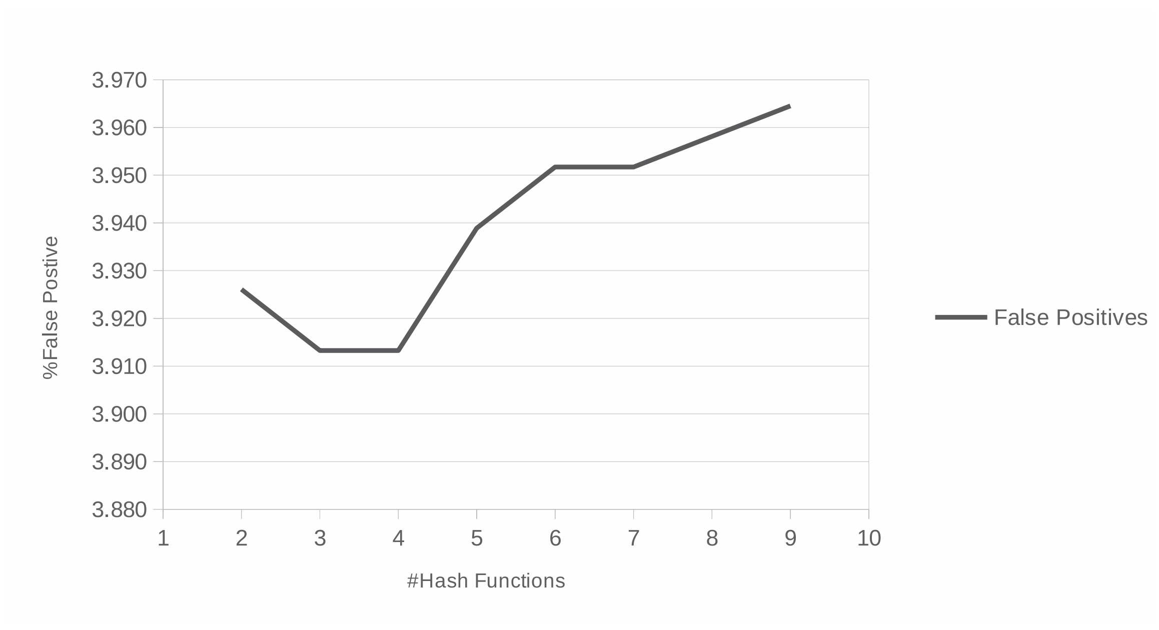


Figure 5.8: False Positive Comparison for Different Number of Hash Functions with Bloom filer Factor 3

Chapter 6
Conclusion

In this thesis, two representations are proposed for the spatial semijoin and their use in multiple-site distributed query processing strategies. To further reduce the data transmission and processing costs, the geometric representation of object MBRs is converted to simple integers or binary bits. The algorithms were evaluated in a real life peer-to-peer distributed system with both synthetic and real data sets. The optimized algorithms perform significantly better than the Distributed Naïve Spatial Semijoin strategy when synthetic data was used. PMSJ is 1.38 times faster than BFSJ in respect to processing cost while BFSJ has a tranmission cost gain of 1.12 over PMSJ. Both the algorithms are 18 times faster and have six times less transmission cost than NSPJ. It is also observed that with the increase of hash functions and Bloom filter factor the false positive percentage rises.

6.1 Future Work

In the future, we are looking towards testing our system against a larger number of distributed nodes. In addition, given the results from the Sequoia 2000 polygon sets, we will consider data distribution in order to lower the false positive rate that occurs with certain data distributions. We are also looking for improved strategies for partitioning the GEMBR space to gain improvements in processing speed. In addition, we will explore more compact representations of the object MBRs for efficient semijoin processing. We are also looking towards implementing some parallel processing strategies to see if these are efficient. Finally, we will test our BFSJ algorithm by using different types of hash functions and number of bloom filters and analyze the performance.

Chapter 7
Appendix

```
Algorithm: distance
Input: lower_coordinate,upper_coordinate
Output: absolute distance between lower and upper coordinates
return absolute(upper_coordinate-lower_coordinate)
```

```
Algorithm: partition_length
Input: lower coordinate, upper_coordinate,
no_of_partitions
#lower and upper coodinates are in the same dimension
Output: length of each partitions
return
  distance(lower_coordinate,upper_coordinate)/no_of_partitions
```

```
Algorithm: partition_function
Input: Global_Encompassing_MBR_Coordinates, no_of_partitions
Output: Coordinates of all respective partitions

partition_length_along_x_axis
<-partition_length(distance(lx,hx),no_of_partitions)
partition_length_along_y_axis
<-partition_length(distance(ly,hy),no_of_partitions)
```

```
#save all the lower and upper coordinates of each partitions in
#partition coodinates array whose index is the number
#of partitions

Algorithm: inside(coordinates_for_object_1,
  coordinates_for_object_2)
Output: true or false
if(lx_of_object_1<=hx_of_object_2
  and lx_of_object_1>=lx_of_object_2)
  and (ly_of_object_1<=hy_of_object_2
  and ly_of_object_1>=ly_of_object_2)
  and (hx_of_object_1<=hx_of_object_2
  and hx_of_object_1>=lx_of_object_2)
  and (hy_of_object_1<=hy_of_object_2
  and hy_of_object_1>=ly_of_object_2)
then
  return true
else
  return false
end if

Algorithm: mapping_indices_and_tuple_ids
Input: all_leaf_node_coordinates,
       global_encompassing_mbr_coordinates,
       no_of_partitions
Output: mapped_indices_and_ids
```

```
partitions_coordinates<-partition_function
  (global_encompassing_mbr,
  no_of_partitions)

for each leaf_node_coordinates belongs to
all_leaf_node_coordinates
 for each partition_coordinates
  if (inside(lower_coordinates_of_leaf_node,
      partitions_coordinates))
  then
      lower_coordinates_for_starting_partition
      <-x_y_coordinates_of_lower_partition
  end if
  if (inside(upper_coordinates_of_leaf_node,
      partitions_coordinates))
  then
upper_coordinates_for_ending_partition
<-x_y_coordinates_of_upper_partition
  end if
 end for
end for

for each partitions_coordinates
 if(inside(partitions_coordinates,focused_partitions
 (lower_coordinates_for_starting_partition,
  upper_coordinates_for_ending_partition))
    mapped_indices_and_ids_vector.push(indices,
    tuple_ids)
 end if
end for

return mapped_indices_and_ids_vector

Algorithm: return_indices_with_multiple_entries
Input: all_indices_file
Output: only indices which are common in all the files

for each i belongs to number_of_indices in all_indices_file
 if (index_element[i+1]==index_element[i])
 then
  value++
  if(value==number_of_sites-1)
  then
   return index_element[i]
  end if
 else
  return -1
  value=0
 end if
end for
```

```
Algorithm: extract_tuple_ids_from_common_indices
Input: common_indices_file,
      mapped_local_indices_and_tuple_ids_file
Output: All the tuple_ids that satisfies join condition

for each elements of common_indices_file
 for each elements of
    mapped_local_indices_and_tuple_ids_file
  if (common_indices_file_index==
      mapped_local_indices_and_tuple_ids_file_index)
   qualified_tuple_ids_vector<- respective_tuple_id
  end if
 end for
end for

return qualified_tuple_ids_vector

Algorithm: extract_leaf_node_MBR_coordinates
Input: rtree_file
Output: leaf_node_MBR_Coordinates_and_IDs
#Extracted Highest(Root) Level Node Informations
#using the command line tools sed and awk.

Algorithm: calculate_local_encompassing_MBR
Input: rtree_file
Output: local_encompassing_MBR
```

```
#Extracted only the root level coordinates using
#command line tools

#each lx,ly,hx,hy is pushed to local_encompassing_MBR
#vector

for each lx belongs to all lx_of_the_root_level_nodes
  local_encompassing_MBR<-min(lx)
end for

for each ly belongs to all ly_of_the_root_level_nodes
  local_encompassing_MBR<-min(ly)
end for

for each hx belongs to all hx_of_the_root_level_nodes
  local_encompassing_MBR<-max(hx)
end for

for each hy belongs to all hy_of_the_root_level_nodes
  local_encompassing_MBR<-min(hy)
end for

return local_encompassing_MBR

Algorithm: global_encompassing_MBR
```

```
Input: concatanated local_encomapassing_MBR_files
Output: global_encompassing_MBR

return calculate_local_encompassing_MBR
  (concatenated local_encomapassing_MBR_files)

Algorithm: file_to_vector
Input: any text file
Output: vector

for each line belongs to text file
 vector<-read each line
end for

return vector

Algorithm: PMSJ_BFSJ_Algorithm
Input: partition_info_file,rtree_file
Output: joined_tuple_ids

#Query Site Processing: send partition_info file
#which contains how many partitions, to all the
#sites parallely and ensure the starting of the process
#at each site at the same time.

send_file_to_local_sites_parallely(partition_info_file)
```

```
#pdsh is used for sending files parallely

#Local Sites Processing:Upon arrival of partition_info
#the partition_based_spatial_join processing starts

local_encompassing_MBR_file
<-calculate_local_encompassing_MBR(rtree_file)

#send local encompassing MBR files from each
#site to Query Site

send_files_to_query_site(local_encompassing_MBR_file)
#scp is used for that

#In the Query Site calculate_global_encompassing_MBR using the
#following way

global_encompassing_MBR_file
<-calculate_global_encompassing_MBR
  (local_encompassing_MBR_files)
send_file_to_local_sites_parallely
(global_encompassing_MBR_file)

#At each local site the following funtion is called and
#output is saved in a file
```

```
all_leaf_node_coordinates_file
<-extract_leaf_node_MBR_coordinates(rtree_file)

all_leaf_node_coordinates
<-file_to_vector(all_leaf_node_coordinates_file)
global_encompassing_mbr
<-file_to_vector(global_encompassing_MBR_file)
no_of_partitions
<-file_to_vector(partition_info_file)

mapped_local_indices_and_tuple_ids_file
<-mapping_indices_and_tuple_ids(all_leaf_node_coordinates,
                       global_encompassing_mbr_coordinates,
                       no_of_partitions)

#send the unique indices of each site to query site

local_unique_indices_file
<-unique_indices_only(mapped_local_indices_and_tuple_ids_file)

#PMSJ specific operations
#send the unique indices only
send_files_to_query_site(local_unique_indices_file)
#find out the common indices from each files of each sites
#at query site
all_indices_file
```

```
<-sort_and_concatenate_file(mapped_local_indices_file)
#used unix command line tools cat and sort -u

common_indices_file
<-return_indices_with_multiple_entries(all_indices_file)
#send common indices to all the sites
send_file_to_local_sites_parallely(common_indices_file)
local_tuple_ids_file
<-extract_tuple_ids_from_common_indices
   (common_indices_file,
  mapped_local_indices_and_tuple_ids_file)

send_files_to_query_site(local_tuple_ids_file)

#BFSJ specific operations
#used unix command line tools for this,sort -u and
#cut to send the unique indices only
bloom_filter_vector
<-bloom_filter_processor
   (local_unique_indices_file,
    no_of_hash_functions,
    no_of_bloom_bits)

send_bloom_filter_to_query_site(bloom_filter_vector)

intersect_bloom_filters(all_bloom_filter_files)
```

```
#all_bloom_filter_files is created by concatenating
#bloom filter
#files from different sites.This function outputs the
#intersected_bloom_filter_file to be sent to local sites.
send_files_to_local_site(intersected_bloom_filter_file)
#parallely sent using pdsh.

bloom_filter_to_index_mapper(intersected_bloom_filter_file)
#This function outputs the file named final_indices_file
#which contains the indices which qualifies to be true for
#certain bloom filter bit position.
local_tuple_ids_file
<-extract_tuple_ids_from_common_indices
    (final_indices_file,
     mapped_local_indices_and_tuple_ids_file)
send_files_to_query_site(local_tuple_ids_file)
```

Bibliography

[1] R. Abdalla and V. Tao. Integrated distributed GIS approach for earthquake disaster modeling and visualization. In *Geo-Information for Disaster Management*, pages 1183–1192. Springer, 2005.

[2] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *IEEE Transactions on Software Engineering*, 9(1):57–68, 1983.

[3] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24th International Conference on Very Large Data Bases*, 1998.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pages 322–331. ACM Press, 1990.

[5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[6] P. Bodorik, J. S. Riordon, and J. S. Pyra. Deciding on correct distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 4(3):253–265, 1992.

[7] T. Brinkhoff, H.-P. Kriegel, and B. 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, New York, NY, USA, 1993. ACM.

[8] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Parallel processing of spatial joins using R-trees. In *Proceedings of the 12th International Conference on Data Engineering*, 1996.

[9] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57, 1984.

[11] T. Hunter. A distributed spatial data library for emergency management. In *Geo-Information for Disaster Management*, pages 733–750. Springer, 2005.

[12] M.-S. Kang, S.-K. Ko, K. Koh, and Y.-C. Choy. A parallel spatial join processing for distributed spatial databases. In *Proceedings of the 5th International Conference on Flexible Query Answering Systems*, pages 212–225, 2002.

[13] O. Karam and F. Petry. Optimizing distributed spatial joins using R-trees. In *Proceedings of the 43rd ACM Southeast Conference*, 2006.

[14] W. Osborn and S. Zaamout. Multiple-site distributed spatial query optimization using spatial semijoins. In *Proceedings of the 10th International Baltic Conference on Databases and Information Systems - Local Proceedings*, pages 11–19, 2012.

[15] M.T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Springer, 2011.

[16] D.J. Maguire D.W. Rhind P.A. Longley, M.F. Goodchild. *Geographical Information Systems Principles, Technical Issues, Management Issues and Applications*. Wiley, 1999.

[17] J.M. Patel and D.J. DeWitt. Partition based spatial-merge join. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 259–270, 1996.

[18] S. Ravada, S. Shekhar, C.-T. Lu, and S. Chawla. Optimizing join index based join processing: A graph partitioning approach. In *SRDS*, pages 302–308, 1998.

[19] H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[20] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice Hall, New Jersey, 2003.

[21] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The SEQUOIA 2000 storage benchmark. *SIGMOD Record*, 22(2), 1993.

[22] K.-L. Tan, B.C. Ooi, and D.J. Abel. Exploiting spatial indexes for semijoin-based join processing in distributed spatial databases. *IEEE Transactions on Knowledge and Data Engineering*, 12(6), 2000.

[23] A. Tripathy, L. Mishra, and P.K. Patra. An efficient approach for distributed spatial query optimization using filters. In *Proccedings of the 3rd Intenational Conference on Advanced Computer Theory and Engineering (ICACTE 2010)*, 2010.

[24] Hua Y, B. Xiao, and J. Wang. Br-tree: A scalable prototype for supporting multiple queries of multidimensional data. *IEEE Transactions on Computers*, 58(12):1585–1598, 2009.

[25] S. Zaamout and W. Osborn. A strategy for optimizing a multi-site query in a distributed spatial database. In *Proceedings of the 12th International Conference on Web and Wireless Geographical Information Systems*, W2GIS'13, pages 16–24, Berlin, Heidelberg, 2013. Springer-Verlag.