Debugging Techniques for Distributed R-Trees

Sávio Teles[1], **Jose Ferreira de S. Filho**[1]

[1]Instituto de Informática – Universidade Federal de Goiás (UFG)
Alameda Palmeiras, Quadra D, Câmpus Samambaia
131 - CEP 74001-970 – Goiânia – GO – Brazil

savioteles@gmail.com, jkairos@gmail.com

Abstract. *R-Trees arrived on the scene in 1984 by Antonin Guttman, as a mechanism of handling spatial data efficiently, building an index structure that helps retrieve data items quickly according to their spatial locations. The R-tree has found significant use in both theoretical and applied contexts. However, it brings a big challenge in terms of debugging. In this paper, we argue that distributed debugging is a non-trivial task and few efforts have been developed for distributed debugging of spatial objects based on R-Tress. We use examples of this issue and briefly discuss basic debugging techniques, then we propose RDebug, a technique for debugging a distributed R-Tree.*

1. Introduction

The Internet has revolutionized the computer and Geographical Information Systems (GIS) like nothing before. In fact the Internet brought big changes how systems store, retrieve and analyze spatial data. The ever-increasing of the large geospatial datasets and the widely application of the complex geocomputation make the parallel processing of GIS an important component of high-performance computing. Thus, spatial distributed applications came on the scene. We define a spatial distributed application as one which software components located at networked computers communicate and coordinate their actions for geoprocessing.

In order to handle spatial data efficiently, a database system needs an index mechanism that will help it retrieve data items quickly according to their spatial locations. An R-tree is a dynamic index structure, which meets this need and is broadly used by in GIS. Generally, the R-Tree index is built and processed in a single machine. However, process an R-Tree in a single machine is not feasible because of the huge size of the geospatial datasets. Thus, many researches such as [An et al. 1999, de Oliveira et al. 2011, Zhong et al. 2012], show that a distributed index structure spanning the workstations can provide an efficient shared storage structure that can be used to gather geographic information more efficiently.

A big challenge though has arisen of spanning the R-Tree index structure among computers: How debug a distributed R-Tree index structure?

Debugging is an essential step in the development process, albeit often neglected in the development of distributed applications due to the fact that distributed systems complicate the already difficult task of debugging.

In recent years, researchers have been developed some helpful debugging techniques for distributed environment. Nevertheless, we have not found any technique to debug a distributed R-Tree. In this paper, we propose a debugging technique to debug the data insertion in an R-Tree, besides of debugging the quality of the index built on the cluster.

The debug method, hereafter called RDebug, uses the distributed index structure to aggregate debugging information. RDebug is used by DistGeo, a shared-nothing platform for distributed spatial algorithms processing. We also created a graphical tool to visualize the debugging information and the R-Tree index structure.

The main achievements of this paper are:

- RDebug - A debugging technique to the data insertion algorithm in a distributed R-Tree.
- DistGeo - A Shared-nothing platform, without point of failure, to process distributed spatial algorithms of an R-Tree.
- A graphical tool to visualize debugging information and the distributed R-Tree index.

The rest of the paper is structured as follows. Section 2 describes the distributed processing of spatial algorithms, Section 3 presents our approach for distributed R-Tree debugging. In Section 4, we briefly give an overview of the use of debugging techniques for distributed environments and the view of the distributed spatial algorithms. Finally, we close the paper with some concluding remarks in Section 5.

2. Distributed Processing of Spatial Algorithms

2.1. Data Structures for Spatial Data Processing

Spatial data objects often cover areas in multi-dimensional spaces and are not well represented by point locations For example, map objects like counties, census tracts etc occupy regions of non-zero size in two dimensions A common operation on spatial data is a search for all objects in an area, for example to find all counties that have land within 20 miles of a particular point. This kind of spatial search occurs frequently in computer aided design (CAD) and geo-data applications, and therefore it is important to be able to retrieve objects efficiently according to their spatial location.

An index based on objects spatial locations is desirable, but classical one-dimensional database indexing structures are not appropriate to multi-dimensional spatial searching structures based on exact matching of values, such as hash tables, are not useful because a range search is required Structures using one-dimensional ordering of key values, such as B-trees [Bayer and McCreight 1970, Bentley 1975b] and ISAM indexes, do not work because the search space is multl-dimensional.

A number of structures have been proposed for handling multi-dimensional point data, such as: KD-Tree [Bentley 1975a], Hilbert R-Tree [Kamel and Faloutsos 1994] and R-Tree [Guttman 1984].

R-Trees were proposed by Antonin Guttman in 1984 and have found significant use in both theoretical and applied contexts, they are tree data structures used for spatial access methods, i.e, for multi-dimensional information such as geographical coordinates, rectangles, polygons. Similar to the B-Tree [Comer 1979], the R-Tree is also a balanced search tree (so all leaf nodes are at the same height).

The key idea of the data structure is to group nearby objects and represent them with their minimum bounding rectangle (MBR) in the next higher level of the tree. Since all objects lie within this bounding rectangle, a query that does not intersect the bounding rectangle also cannot intersect any of the contained objects. At the leaf level, each rectangle describes a single object; at higher levels the aggregation of an increasing number of objects. This can also be seen as an increasingly coarse approximation of the data set.

Figure 1 illustrates the hierarchical structure of an R-Tree with a root node, internal nodes (N1...2 C N3...6) and leaves (N3...6 C a...i). The Upper Top of Figure 1 shows MBRs grouping spatial objects of a..a i in sets by their co-location. The bottom of Figure 1 illustrates the R-Tree representation. Each node stores at most M and at least m <=M/2 entries [Guttman 1984].

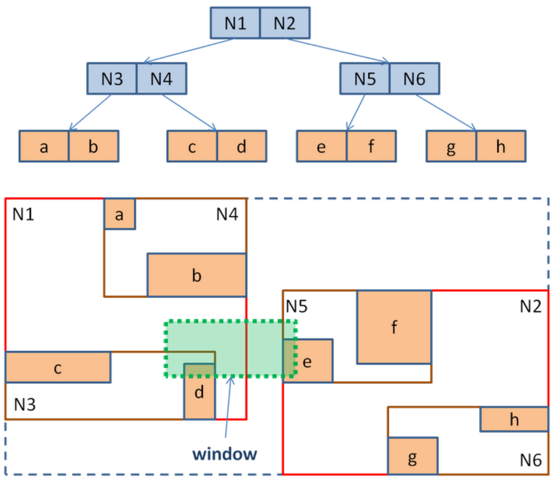

Figura 1. R-Tree Structure

The input is a search rectangle (Query box). Searching is quite similar to searching in a B+ tree. The search starts from the root node of the tree. Every internal node contains a set of rectangles and pointers to the corresponding child node and every leaf node contains the rectangles of spatial objects (the pointer to some spatial object can be there). For every rectangle in a node, it has to be decided if it overlaps the search rectangle or not. If yes, the corresponding child node has to be searched also. Searching is done in a recursively until all overlapping nodes have been traversed. When a leaf node is reached, the contained bounding boxes (rectangles) are tested against the search rectangle and their objects (if there are any) are put into the result set if they lie within the search rectangle.

Dead space is a space which is indexed but does not contain data. In Figure 1, N1 area is an example of dead space. Dead spaces cause the search go into false sub-trees. In figure 1, window K represents the intersection between the MBR dead space and N2, the search walks through the sub-tree of N2, although this sub-tree does not contain any data to return. Overlapping areas are regions of intersection between polygons. The area between N3 and N4 in Figure 1 is an example of overlapping. Less overlapping reduces the amount of sub-trees accessed during r-tree traversal. The overlapping area between N3 and N4 in Figure 1, forces the traversal of the two sub-trees, degrading the performance of R-Tree [Beckmann et al. 1990].

2.2. DistGeo: A Platform of Distributed Spatial Operations for Geoprocessing

There are two important requirements for processing data in shared-nothing (cluster) [DeWitt and Gray 1992] architectures, First divide the data in partitions. The second requirement is distribute this partitions in the cluster nodes. For spatial datasets, we use the same idea of partitions, and the partitions distribution considers the geographic co-location among the polygons to speed up the searching algorithm.

Figure 2 illustrates the structure of a Distributed R-Tree in a cluster. The partitioning it is performed grouping the nodes in cluster and creating the indexes according to the R-Tree structure. The lines in Figure 2 show the need for message exchange to reach the sub-trees during the algorithm processing.

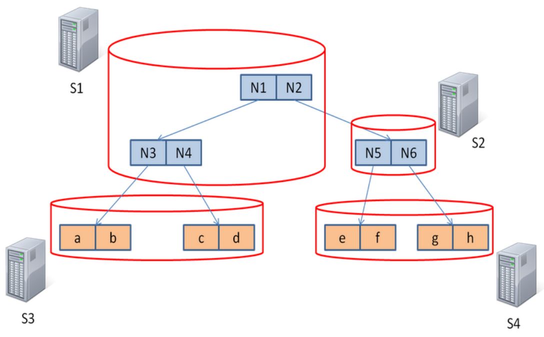

Figura 2. R-Tree Partitioning in a shared-nothing architecture

Insertions and searching in a distributed R-Tree are similar to the non-distributed version, except for: i) The need of message exchange to access the distributed partitions ii) Concurrency control and consistency due to the parallel processing in the cluster. The data distribution among the distributed partitions in the cluster is both the main factor to the parallel processing and also the main challenge when building the distributed index. Quality of index is another factor, which influences the communication. An index with high quality reduces the searching space and the sub-tress access leading to less number of message exchange. This section describes the main architecture decisions and some implementation details that molds the challenges when constructing a solution for distributed Geo-processing. The distributed index has been built according to the taxonomy defined in [An et al. 1999], as follows: i) Allocation Unit: block - A partition is created for every R-Tree node; ii) Allocation Frequency: overflow - In the insert process, new partitions are created when a node in the tree needs to split; iii) Distribution Policy: balanced - To keep the tree balanced the partitions are distributed among the cluster nodes.

DistGeo is based on the shared-nothing architecture, which the nodes do not share CPU, hard disk and memory and the communication relies on message exchange. Figure 3 depicts DistGeo plataform based on peer-to-peer model, with the data manage by the cluster presented as a ring topology. It is divided in ranges of keys, which are managed for each node of the cluster. To a node join the ring it must first receive a range The range of keys are known by each node in the cluster. For instance, in a ring representation, whose key set start with 0 till 100, if we have 4 nodes in the cluster, the division could be done as shown below: a) 0-25, b) 25-50, c) 50-75 e d) 75-100. If we want to search for one object with key 34, we certainly should look on the node 2.

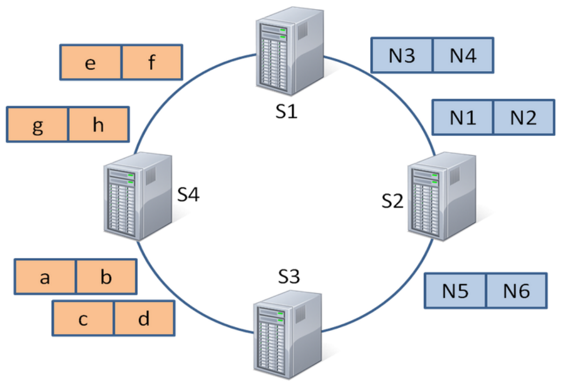

Figura 3. Figure 3

Reliability and fault-tolerance are implemented storing the R-Tree nodes in multiple nodes in the cluster. Each node N receives a key, which is used to store the node in a server S responsible for ring range, replicating the node N to the next two servers in S (clockwise).

Data replication is equally important. If a message is sent to a bode N, in the moment of R-Tree traversal an active server is elected, and replicates the data in this node to further requisition processing.

DistGeo utilizes the Gossip protocol, which every cluster node exchanges information among themselves for service discovery and knowing the statuses of the cluster's nodes. In the Gossip protocol every second a message is exchanged among three nodes in the cluster, consequently every cluster's node have knowledge of each other.

Read/Write operations may be performed in any node of the cluster. When a request is made to a cluster's node, it becomes the coordinator of the operation requested by the client. The coordinator works as a proxy between the client and the cluster's nodes. In a distributed R-Tree, the requests are always sent to the cluster's node that stores the root node of the R-tree.

3. A Technique for Debugging A Distributed R-Tree

Spatial index debugging is a big challenge in a distributed R-Tree and this section describes a new technique RDebug, which allows debugging the index building of an R-Tree.

The R-tree index building follows a top-down approach, in other words, the index is always built from root to leaves. Debugging the index reliability as the index is built is a non-trivial task, and the aim of this paper is to show a technique for index debugging after it has been created. Common challenges when working with an R-Tree are: i) Reliability of the nodes replicas of the R-Tree, ii) ensure that the MBR of the parent nodes intersect the MBR of their children, iii) the existence of duplicated nodes or being referenced by more than one parent node, and iv) if the value M and m of the nodes are compliant with the R-Tree descriptions as shown in Section 2. Furthermore, it is possible to access index data to help in its optimization as dead space and overlapping area.

Algorithm 1 shows the RDebug technique for debugging the distributed spatial index, using the index structure itself. The algorithm has two steps: 1) The algorithm processing is similar to the search in an R-Tree; 2) The algorithm does the inverse of a search in an R-Tree appending information to the distributed index.

The first step, called S1 [Search sub-trees] (lines 1 - 11), the Algorithm 1 traverses every node of the R-Tree starting from the root node to the leaves. Its purpose is to spread the debugging algorithm. The first request is sent to any server, which stores a replica of the root node.

If the node T is not a leaf (lines 2 - 8), then the number of children entries is stored to control the number of expected answers to this node in the second step of the algorithm. This information is stored in a shared memory accessed by all servers with a replica of T. Lines 4 -7, show that for every entry E in the node, a message is sent (continuing step S1) to any server that holds a replica of the child node of E, carrying on the first step in the children nodes. If the node is a leaf, the second step, named S2 [Aggregation] is started.

Second step aim (lines 12 - 41) is to aggregate the information used for future debugging. This step receives the debugging information of every child node of T. Therefore, for a given node T with n children, the second step is invoked n times in the node T.

The index itself is used to aggregate this information, the computational resources of the cluster helps improve the debugging information aggregation time. The index reverse structure allows, besides of spanning the aggregation information processing, build the debugging aggregation information, as one node of the R-Tree is responsible to aggregate only the information of its children.

Line 13 verifies the consistency of T in the servers that store any replica of T. Line 14 verifies the consistency of M and m. Lines 16 and 17 calculate the dead space and overlapping area for each node of the R-tree. Those information help the insertion algorithm designer analyze the quality of the built index. Beside this, lines 18 - 22, we have information of the MBRs of each node for every entry of the node. This information can be used as an input to a tool capable of visualizing the index of the R-Tree.

If the aggregation step is being executed in the leaves, then if T is the root node (lines 24 - 26), the node information are sent to the client application. IF T is not the root node, in line 27, the information are sent to the parent node of T. If the aggregation step is in an internal node (lines 29 - 43), the algorithm aggregates the information of the children nodes. Line 29, the algorithm receives the information sent by the child node. Line 30, verifies if the MBR of the entry that points to the child node is indeed the same MBR sent by the child node.

The information of the children nodes are stored in a shared memory, with concurrency control, by the replicas of T. Hence, line 31, those information are accessed from the shared memory. Line 32, adds the data processed from lines 29 and 30. Line 33, acquires the number of children nodes that sent debugging information in the shared memory of replicas T. This information is stored in the variable $count$ and $count$ is decremented to let the other replicas know.

If every node has sent the answer, the variable count then will hold the value 0 and lines 35-39 are processed. If T is the root node, then the information are sent to the client application, otherwise, those information are sent to the parent node of T. If the variable $count$ is greater than 0, then the client information are stored in the shared memory.

Algoritmo 1: $RDebug(T)$

```
Entrada: $T$ reference to root node of R-Tree $tree$
Saída: Debugging information about distributed R-Tree $tree$
S1 [Search subtrees]
if $T$ is not leaf then
    store the number of child entries in each replica server of T
    for each entry $E$ in $T$ do
        $server$ ← choose one server, randomically, that store one replica of $E$
        send msg to $server$ to process the node child of $E$ on step S1
    end
else
    verifiy the consistency of $T$ in others replicas
    Invoke step S2 [Aggregation]
end
S2 [Aggregation]
$replica\_consistency$ ⇐ verifiy the consistency of $T$ in others replicas
$node\_consistency$ ⇐ verify the consistency of $M$ and $m$ values of $T$
add in $informations$: $replica\_consistency$ and $node\_consistency$
$overlap$ ⇐ overlap area of $T$
$dead\_area$ ⇐ dead area of $T$
$bound$ ⇐ MBR of $T$
$list$ ⇐ $\emptyset$
for for each entry $E$ in $T$ do
    add the $MBR$ of $E$ in $list$
end
if $T$ is leaf then
    if $T$ is root then
        send response with R-Tree nodes information to app client
    end
    send msg with $informations$ to parent of $T$
else
    $entry\_info$ ⇐ information sent by child node
    $mbr\_consistent$ ⇐ verify if the bound of the child node is equal to
    bound of entry of T that points to this child
    $informations$ ⇐ the child information stored on shared memory by
    replicas of $T$
    add in $informations$: $entries\_info$, and $mbr\_consistent$
    $count$ ⇐ retrieve the number of entries child which not sent a debugging
    response and decrement by 1 unit
    if count == 0 then
        if $T$ is root then
            send response with $informations$ to app client
        else
            send msg with $informations$ to parent of $T$
        end
    else
        store $informations$ on shared memory
    end
end
```

The algorithm 1 was implemented in the DistGeo platform to collect the debugging information of the built distributed R-tree. Those information are used in the platform to find out indexing issues and to optimize the R-tree index for searching. Figure 4.1, shows a graphical tool created to visualize the structure of the distributed R-Tree index, using as the input the information generated by the distributed debugging algorithm in DistGeo platform.

[Colocar a Figura aqui!!]

With the aid of RDebug 1 algorithm, it is possible debug the searching algorithms of an R-Tree. E.g: The Window Query algorithm shown on Section 2.1. To tweak RDebug to Window Query, it is only needed add an window query in the first step and gather the aggregation information of the accessed nodes. Whereas, the algorithms that access diverse R-Trees, such as Spatial Join, need a deep change, as the algorithms can go through different paths.

4. Related work

Several techniques exist for debugging distributed, concurrent, and parallel programs. Although none of the existing approaches provide support for debugging a distributed R-Tree index [Manolopoulos et al. 2003, Jacox and Samet 2007]. Along with our discussion, we highlight the techniques that influenced the design of RDenug, a new algorithm for debugging a distributed R-Tree index and a peer-to-peer platform to process distributed spatial algorithms.

Researches on distributed spatial data either show techniques to debug distributed applications in general or techniques for R-tree distributed processing. The Section 4.1 shows the distributed debugging researches and 4.2 describes researches of platforms for processing distributed spatial algorithms.

4.1. Distributed Debugging Techniques

In [Boix and Hondt 2011] the author breaks down debuggers in two main families: log-based debuggers (also known as post-mortem debuggers) and breakpoint-based debuggers (also known as online debuggers). Log-based debuggers insert log statements in the code of the program to be able to generate a trace log during its execution. Breakpoint-based debuggers, on the other hand, execute the program in the debug mode that allows programs to pause/resume the program execution at certain points, inspect program state, and perform step-by-step execution. [WH Cheung 1990] describes a process for debugging in general and does not focus on a specific debugger or a particular technique.

«@TODO: TO BE CONTINUED - MISSING CONTENT - CONTINUE FROM HERE»

4.2. Distributed Spatial Algorithms

This Section describes briefly the researches that present the use of parallelism in order to improve the response time of the spatial algorithms. M-RTree [Koudas et al. 1996] was the first published paper, which shows a shared-nothing architecture, with a master and several workstations connected to a LAN. The master machine handles a high volume of computation, besides of processing some directories of the R-Tree, it merges the answers to the client machines. Same technique is found on MC-RTree [Schnitzer and Leutenegger 1999].

In [An et al. 1999], it is described that a Network of Workstations (NOW) - also referred to as cluster systems - is a cost-effective solution for high performance. The paper proposes an architecture similar to M-RTree and MC-RTree, with the same disadvantages of both.

In SD-RTree [du Mouza et al. 2007], a binary tree is used instead of an R-Tree. The binary tree increases the number of messages, since the data representation in a binary tree has more levels than the same data represented in an R-Tree. Hadoop-GIS [Kerr 2009] shows a scalable and high performance spatial data warehousing system for running large scale spatial queries on Hadoop. However the gain running large scale queries, it does not use indexing to improve the performance of the operations in the datasets.

[de Oliveira et al. 2011] presents a distributed platform for spatial operations. Although, the solution proposed implements a distributed index, it is not scalable, since every message go through the replicated master node. [saviosbrc] shows a hybrid peer-to-peer platform to process the distributed spatial joint. The architecture presented in [saviosbrc] comprehends a set of machines for naming resolution and events management. Thus, these machines could be a bottleneck in the system.

[Xie et al. 2008] introduces a two-phase load-balancing scheme for the parallel GIS operations in distributed environment.

[Zhang et al. 2009] describes MapReduce and shows how spatial queries can be naturally expressed in this model, without explicitly addressing any of the details of parallelization. Although the geocomputation high performance with this approach, it is only indicated for non-indexed datasets.

In [Zhong et al. 2012], an approach is proposed for "indexing + MapReduce"data processing architecture to improve the computation capability of spatial query .The spatial index is distributed in two levels: locally and globally.

A number of techniques and platforms have been proposed for handling big spatial data, nevertheless none of them propose a platform using a peer-to-peer approach for processing distributed spatial algorithms as found on DistGeo platform. Besides, none of the researches propose a technique for distributed spatial index debugging of an R-Tree.

5. Conclusion

DistGeo platform presents an approach for processing the distributed spatial operations through the distributed R-Tree index. Due to the distributed processing nature on this platform an issue arises: Debugging the insertion algorithm when the R-tree nodes a distributed in a cluster.

We have seen researches on spatial data processing and distributed debugging, but none of them propose techniques for debugging the spatial algorithms of an R-Tree. In this paper, we present RDebug, a technique for debugging the distributed index building of an R-Tree. RDebug, uses the R-Tree index itself to gather the debug information. The information gathering is done in the R-Tree index using a down-top traversal in the tree.

Utilizing the distributed index itself, the data gathering can be achieve in a distributed way, improving the debugging algorithm efficiency.

RDebug, has been implemented in DistGeo platform. The R-Tree nodes are distributed and replicated over the cluster. Thus, RDebug can be processed without bottlenecks and point of failures. Besides, the R-Tree replicated nodes in the cluster allow load-balancing in the distributed R-Tree index traversal. During the traversal, at every node access of the R-Tree, the traversal might go to a node of the cluster with less workload increasing RDebug performance. The information exchange of the machines statuses is done trough the Gossip algorithm.

A graphical tool has been created to visualize the structure of the distributed R-Tree index and the debugging information about the index building. With these input we can identify discrepancies in the index building and optimize it.

Ongoing work includes modify the RDebug algorithm to debug the Window Query and Join Query searching algorithms. The RDebug algorithm is easily adapted to gather debugging information for Window Query. Whereas, for Join Query algorithm, RDebug must be changed considerably, since the traversal is processed in two different distributed R-Trees. Another ongoing work is to realize performance tests of the RDebug algorithm in DistGeo platform.

Referências

An, N., Lu, R., Qian, L., Sivasubramaniam, A., and Keefe, T. (1999). Storing spatial data on a network of workstations. *Cluster Computing*, 2(4):259–270.

Bayer, R. and McCreight, E. (1970). Organization and Maintenance of Large Ordered Indices. *ACM-SIGFTDET Workshop on Data Description and Access*, pages 107–141.

Beckmann, N., Kriegel, H., Schneider, R., and Seeger, B. (1990). *The R*-tree: an efficient and robust access method for points and rectangles*, volume 19. ACM.

Bentley, J. (1975a). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):517.

Bentley, J. L. (1975b). Multidimencional Binary Search Trees Used for Associative Searching. *ACM*, pages 509–517.

Boix, E. G., C. T. V. N. C. M. W. D. and Hondt, T. D. (2011). REME-D: a Reflective Epidemic Message-Oriented Debugger for Ambient-Oriented Applications. *ACM*, pages 1275–1281.

Comer, D. (1979). Ubiquitous B-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137.

de Oliveira, T., Sacramento, V., Oliveira, S., Albuquerque, P., Cardoso, M., Bloco, I., and Campus, I. (2011). DSI-Rtree - Um Índice R-Tree Escalável Distribuído. In *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*.

du Mouza, C., Litwin, W., and Rigaux, P. (2007). Sd-rtree: A scalable distributed rtree. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 296–305. IEEE.

Guttman, A. (1984). *R-trees: a dynamic index structure for spatial searching*, volume 14. ACM.

Jacox, E. and Samet, H. (2007). Spatial join techniques. *ACM Transactions on Database Systems (TODS)*, 32(1):7.

Kamel, I. and Faloutsos, C. (1994). Hilbert R-tree: An Improved R-tree using Fractals. In *VLDB 20th*, page 509. Morgan Kaufmann Publishers Inc.

Kerr, N. (2009). Alternative Approaches to Parallel GIS Processing. *Arizona State University - Master Thesis*.

Koudas, N., Faloutsos, C., and Kamel, I. (1996). Declustering spatial databases on a multi-computer architecture. *Advances in Database Technology-EDBT'96*, pages 592–614.

Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A., and Theodoridis, Y. (2003). R-trees have grown everywhere. *Submitted to ACM Computing Surveys*.

Schnitzer, B. and Leutenegger, S. (1999). Master-client r-trees: A new parallel r-tree architecture. In *Scientific and Statistical Database Management, 1999. Eleventh International Conference on*, pages 68–77. IEEE.

WH Cheung, JP Black, E. M. (1990). A Framework for Distributed Debugging. *IEEE*, pages 106–115.

Xie, Z., Ye, Z., and Wu, L. (2008). A two-phase load-balancing framework of parallel gis operations. In *Geoscience and Remote Sensing Symposium, 2008. IGARSS 2008. IEEE International*, volume 2, pages II–1286. IEEE.

Zhang, S., Han, J., Liu, Z., Wang, K., and Feng, S. (2009). Spatial queries evaluation with mapreduce. In *Grid and Cooperative Computing, 2009. GCC'09. Eighth International Conference on*, pages 287–292. IEEE.

Zhong, Y., Han, J., Zhang, T., Li, Z., Fang, J., and Chen, G. (2012). Towards parallel spatial query processing for big spatial data. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 2085–2094. IEEE.