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. *This meta-paper describes the style to be used in articles and short papers for SBC conferences. For papers in English, you should add just an abstract while for the papers in Portuguese, we also ask for an abstract in Portuguese ("resumo"). In both cases, abstracts should not have more than 10 lines and must be in the first page of the paper.*

1. Introduction

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.

Few techniques have been developed for debugging distributed spatial data; furthermore, no precise or elegant method has been developed for debugging distributed spatial data based on R-Trees data structures.

R-Trees are the most used data structures for spatial data access in a distributed environment, however searching algorithms and debugging is a non-trivial task as the R-Tree nodes are usually distributed in multiple nodes for efficient computation.

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.

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

2. Basic Techniques

In REME-D: a Reflective epidemic Message-Oriented Debugger for Ambient Oriented Applications paper the author breaks down debuggers in two main families (Section 2 - Related Work): 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.

2.1. Log-based Debuggers

Log-based debuggers insert log statements in the code of the program to able to generate a trace log during its execution. It is the most primitive debugging technique, but it is also the easiest to implement. Indeed, it is often the only technique available, as pointed out by Philip Zeyligger of Cloudera during a Tech Talk in 2013 about Tricks for Distributed Debugging and Diagnosis. In this technique, you insert debugging probes, usually output statements, at carefully selected places in the program. Using output data, you try to understand the execution behavior so you can find bugs. The advantage of this technique are that only simple output statements are required and that you see only the data you select.

However, it has disadvantages. First, you must observe the output of processes on multiple processors at the same time. When the number of processes is large, such observation becomes infeasible. Also, the technique relies completely on your ability to select appropriate places in the program to insert output statements - an inexact art greatly dependent on your experience and thinking. Furthermore, the technique requires modifications to the program and thus may alter the program structure or even introduce new bugs.

Despite its disadvantages, combining output debugging with other techniques can make debugging easier.

2.2. Breakpoint-based Debuggers

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. Stack traces, for example, give the developer an idea of what has happened before in the execution of the program, giving hints of how the developer got the current point in the execution. Despite the fact that the stack view does not show total causality, in most cases tracing back the stack is enough to find the cause of the bug. When this does not uncover the cause, breakpoints make it easier to mark to interesting places in the execution.

A breakpoint is a point in the execution flow where normal execution is suspended and cluster-state information is saved. At a breakpoint, you can interactively examine and modify parts of cluster states, like execution status and data values, or control later execution by requesting single-step execution or setting further breakpoints. If you request it, execution will continue after the breakpoint.

The technique requires no extra code in the program, so it avoids some of the unwanted effects of adding debugging probes to distributed programs. Also, you can control cluster execution and select display information interactively. The main disadvantage is that you must be knowledgeable enough to set breakpoints at appropriate places in the program and to examine relevant data. Using breakpoints in a distributed environment raises three problems.

First, it is impossible to define a breakpoint in terms of precise global states. Thus, researchers usually define a breakpoint in terms of events in a process or interactions among processes.

Second, the semantics of single-step execution are no longer obvious. Some peo-

ple define it to be the execution of a single machine instruction or a statement of source code on a local processor. Others consider it to be a single statement on each processor involved. Still others treat an event as a single-step. Executing a single instruction may not be very productive. To find bugs that result from the interaction of processes, it is more effective to run the cluster until a significant event occurs.

Third, there is the problem of halting the process cluster at a breakpoint or after a single step. When a breakpoint is triggered, the whole cluster must be stopped. One simple way to do so is to broadcast pause messages to all processors. A processor completely suspends its execution when it receives the pause message. But in may cases, you want only to stop the process cluster you are interested in, not the entire system. In such cases, you would send out pause messages with a cluster identification. When a processor receives such a message, it suspends only the execution of that cluster's processes. To resume the execution, a continue message is broadcast to all processors.

However, the pause and continue operations are not so simple as they might seem at first. For example, when a process that is subject to a time-out request halts for debugging, the real-time clock is still running. When the process resumes execution, it will encounter a much shorter time-out interval, and its behavior may change significantly. Cooper introduced a logical clock mechanism to maintain correct time-out intervals and this to provide transparent halting [1].

Pause and continue operations also involve the problem of how to halt the cluster in a consistent global state. Chandy and Lamport introduced a distributed algorithm to obtain distributed snapshots of a cluster[2]. The algorithm is intended to capture only global states with stable properties that, once they become true, remain true thereafter (like deadlock and process death).

The algorithm is divided into two independent phases. During the first phase, local state information is recorded at each process. This phase ends when it is determined that all information - both within processes and in transit over communication channels - has been taken into account. In the second phase, the local state information of each process is collected into a snapshot by the processes that initiated the snapshot.

There are two important issues in the use of this algorithm. First, halting a process depends on the interactions between that process and other processes. The algorithm cannot collect local information about a process that has one or no communication channels connected to other processes. Furthermore, consider a process that has only infrequent interactions with other processes: That process would halt long after all other processes have halted. Miller and Choi dealt with this issue by introducing two additional control channels between a debugger process and each process in the cluster [3]. The second issue is that messages must be received in the order in which they are sent. Thus, the algorithm cannot be based directly on a datagram-type communication facility.

3. Distributed Processing of Spatial Algorithms

3.1. Data Structures for Spatial Data Processing

Over the last decades spatial data indexing techniques have been the subject of many researches, which originated diverse data structures, such as: KD-Tree [Bentley 1975], 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; the ”R” in R-tree is for rectangle. 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.

R*-Trees are a variant of R-Trees used for indexing spatial information. R*-trees have slightly higher construction cost than standard R-trees, as the data may need to be reinserted; but the resulting tree will usually have a better query performance. Like the standard R-tree, it can store both point and spatial data. It was proposed by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger in 1990. R*-Trees are the data structure described in this paper. 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].

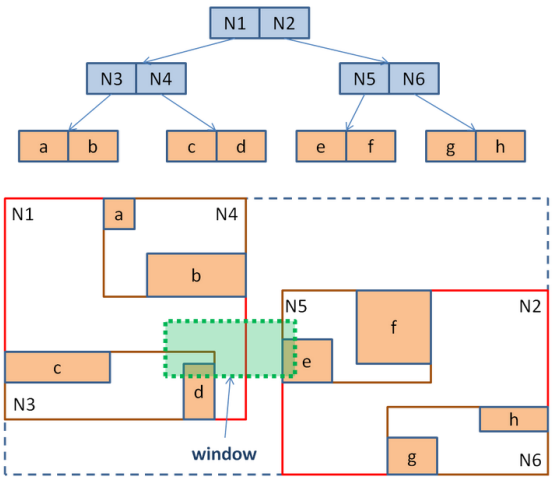

Figure 1. R-Tree Structure

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 [Guttman 1984, Beckmann et al. 1990].

3.2. 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.

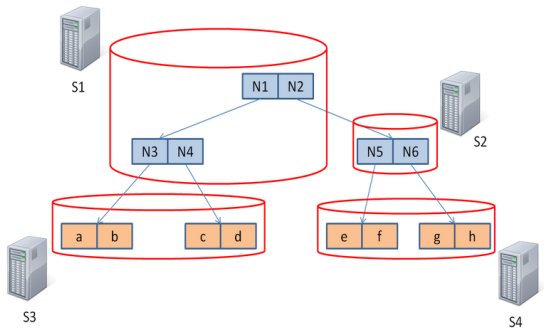

Figure 2. R-Tree Partitioning in a shared-nothing architecture

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 would be done as shown below:

1. node 1: 0-25
2. node 2: 25-50
3. node 3: 50-75
4. node 4: 75-100

If we want to search for one object with key 34, we certainly should look on the node 2.

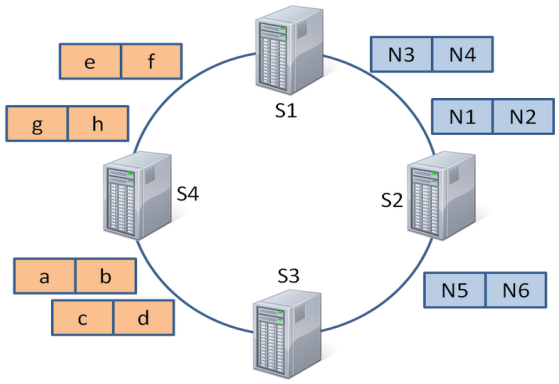

Figure 3. Figure 3

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.

The Gossip protocol, which every cluster node exchanges information among themselves is used by DistGeo for service discovery and knowing the status of the clus-

ter's nodes. 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 cluster's node, which stores the root node of the R-tree.

4. A Technique for Debugging A Distributed R-Tree

Spatial index debugging is a big challenge in a distributed R-Tree and this sections describes a new technique, known as RDebug, which allows debugging the index creation 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 3,1. Furthermore, it is possible to access index data to help in its optimization as dead space and overlapping area.

Table 1 shows the RDebug technique for debugging the distributed spatial index, using the index structure itself. The algorithm has two phases:

1) The algorithm processing is similar to the search in an R-Tree;

2) The algorithm does int inverse of a search in an R-Tree appending information to the distributed index.

Listing 1. RDebug Algorithm

```
Given an R-tree whose node is namely T, the algorithm
starts on root node

S1 [Search subtrees]
If T is not leaf:
        store the number of entries child
        verifiy the consistency of T in others replicas
        verify the consistency of M and m values

        for each entry E in T
                server <- choose one server, randomically,
that store one replica
                                of E
                send msg to server to process the node T
                child on step S1

If T is leaf
        verifiy the consistency of T in others replicas
        Invoke step S2 [Aggregation]

S2 [Aggregation]
overlap <- calculate overlap area of T
dead_area <- calculate dead area of T
bound <- get the MBR of T
list = {}     //contains the MBRs of entries of T
```

```
for each entry E in T
        add the MBR of E in the list

If T is not leaf:
    entries_info <- informations of the entries sent by
                                child node
    bound_child <- bound of the child node
    verify if the bound of the child node is equal to bound
    of entry of T that points to this child
    count <- retrieve the number of entries child and
    decrement by 1 unit
    if(count == 0)
        if T has no parent
            send response with R-Tree nodes information to
            app client
        else
            send msg with child informations to parent node
else
     if T has no parent
            send response with R-Tree nodes information to
            app client
     else
            send msg with child informations to parent node
```

The First Step called of S1 [Search sub-trees], the algorithm traversals every R-Tree node starting from the root node to the leaves. This step objective, beside sparse the debugging algorithm, it is identify consistency issues in the R-Tree nodes and the M and m value consistency.

The first requisition is sent to a server that stores a replica of the root node, the consistency of the replicas nodes and the M an m values are verified for every node. If the current node T is not a leaf then the number of children is stored to control the number of wanted answers in the second step of the algorithm. For each entry E in the node, a message is sent (to process the step S1) to any server, which contains the replica of the child node of E. If the current node is a leaf, the second step, known as S2 [Aggregation] begins.

5. Conclusion

6. References

Bibliographic references must be unambiguous and uniform. We recommend giving the author names references in brackets, e.g. [**?**], [**?**], and [**?**].

The references must be listed using 12 point font size, with 6 points of space before each reference. The first line of each reference should not be indented, while the subsequent should be indented by 0.5 cm.