Debugging Techniques for Distributed R-Trees

So Teles[1], Jose Ferreira de S. Filho[1]

[1]Instituto de Informca – Universidade Federal de GoiUFG)
Alameda Palmeiras, Quadra D, Cus Samambaia
131 - CEP 74001-970 – Goia – 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 a technique for debugging a distributed R-Tree.*

1. Introduction

A popularização da Internet trouxe grandes mudanças nos sistemas armazenam, recuperam e analisam dados referentes ao mundo geográfico. Devido a crescente disponibilidade de dados e aumento no número de usuários, houve a necessidade de distribuir os dados destas aplicações entre vários computadores. Com isto emergiram as aplicações espaciais distribuídas, que podem ser definidos como um conjunto de computadores interconectados por uma rede de computadores que cooperam para a realização de geoprocessamento nas bases de dados disponíveis no sistema.

Estas aplicações utilizam o índice espacial R-Tree para indexar os dados armazenados. Geralmente, as aplicações controem e processam o índice R-Tree de forma centralizada. Entretanto, para grandes quantidade de dados e usuários torna-se impossível processar os algoritmos da R-Tree de forma eficiente em apenas uma máquina. Por isso, diversos trabalhos, tais como [An et al. 1999, de Oliveira et al. 2011, Zhong et al. 2012], distribuem os nós do índice R-Tree entre várias máquinas de um cluster para que os algoritmos sejam processados de forma distribuída.

A distribuição dos nós da R-Tree no cluster ocasionou o surgimento de um desafio: como realizar a depuração dos algoritmos espaciais em um cluster de computadores. 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. Por isso, este trabalho tem como objetivo propor uma técnica para depuração do algoritmo de inserção de dados da R-Tree, além de depurar a qualidade do índice distribuído construído no cluster.

O método de depuração proposto, denominado RDebug, utiliza a própria estrutura distribuída do índice para agregar as informações de depuração. Para testar este algoritmo, a plataforma shared-nothing de processamento de algoritmos espaciais distribuídos, DistGeo, foi construída para implementar os algoritmos distribuídos da R-Tree e o algoritmo de depuração RDebug. Uma aplicação gráfica foi construída para visualizar as informações de depuração e a estrutura do índice R-Tree.

As principais contribuições deste trabalho são:

- Proposta de um algoritmo para depuração do algoritmo de inserção de dados na R-Tree distribuída.
- A implementação de uma plataforma shared-nothing, sem pontos de falhas, para o processamento dos algoritmos espaciais distribuídos da R-Tree.
- Implementação de uma aplicação gráfica para visualização das informações de depuração e do índice R-Tree distribuído.

The rest of the paper is structured as follows. In Section 2 describes the distributed processing of spatial algorithms. The Section 3 describes 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 ii 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 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 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. 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].

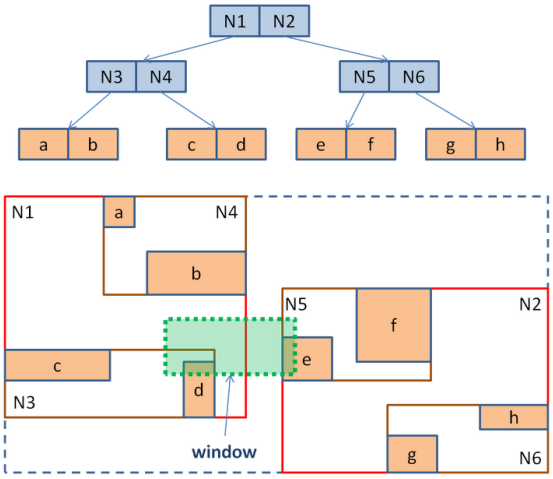

Figura 1. R-Tree Structure

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.

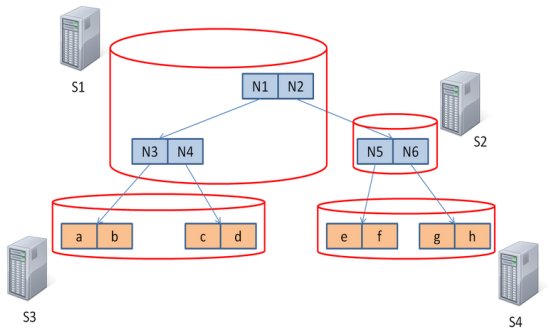

Figura 2. R-Tree Partitioning in a shared-nothing architecture

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 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.

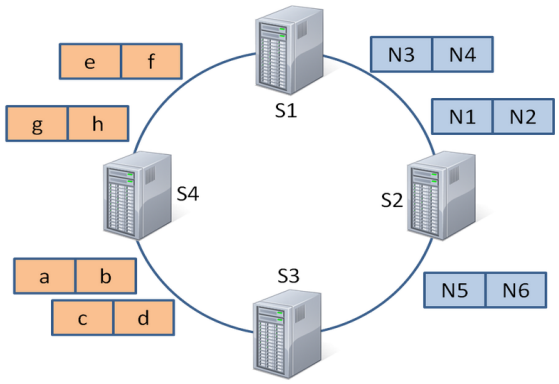

Figura 3. Figure 3

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.

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

3. 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 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 int inverse of a search in an R-Tree appending information to the distributed index.

Na primeira etapa, denominada S1 [Search subtrees] (linhas 1 a 11), o Algoritmo 1 percorre todos os nós da R-Tree a partir do nó raiz até chegar nas folhas. Esta primeira etapa têm como objetivo espalhar o algoritmo de debugging. A requisição inicial é enviada para algum servidor que armazena uma réplica do nó raiz.

Se o nó T analisado não for folha (linhas 2 a 8), então é armazenado o número de entradas filhas para que seja utilizado para controlar o número de respostas esperadas para este nó na segunda etapa do algoritmo. Este valor é armazenado em uma memória compartilhada por todas os servidores que tem alguma réplica de T. Nas linhas 4 a 7, para cada entrada E no nó, uma mensagem é enviada (continuando a etapa S1) para algum servidor que contém uma réplica do nó filho de E, continuando assim a primeira etapa nos nós filhos. Se for folha, a segunda etapa, denominada S2 [Aggregation] , é iniciada.

O objetivo da segunda etapa (linhas 12 a 41) é agregar os valores utilizados para depuração posteriormente. Esta segunda etapa recebe as informações de depuração de cada nó filho de T. Portanto, para um determinado nó T com n filhos, a segunda etapa é chamada n vezes no nó T.

Utilizando o próprio índice para agregar estes valores, pode-se utilizar os recursos computacionais disponíveis no cluster para minimizar o tempo de agregação das informações de depuração. A utilização da estrutura reversa do índice permite, além de distribuir o processamento da agregação de informações, facilitar a construção do código de agregação de informações de depuração, já que um nó da R-Tree é responsável por agregar as informações apenas de seus filhos.

Na linha 13 é verificado a consistência de T nos servidores que armazenam alguma réplica de T. Na linha 14, são verificado a consistência dos valores de M e m. Nas linhas 16 e 17 são calculados a área morta e área de sobreposição em cada nó da R-tree. Estes valores irão permitir que o projetista do algoritmo de inserção da R-Tree possa analisar a qualidade do índice construído. Além destes valores, nas linhas 18 a 22, são obtidos os MBRs dos nós e de cada entrada do nó. Estes valores podem ser utilizados por uma ferramenta para visualizar a estrutura do índice R-Tree.

Se a etapa de agregação estiver sendo executada nas folhas, então se T for raiz (linhas 24 a 26), as informações do nó são enviadas para a aplicação cliente. Se T não for raiz então, na linha 27, as informações são enviadas para o nó pai de T. Se a etapa de agregação estiver em um nó interno (linhas 29 a 43), o algoritmo deve agregar as informações dos nós filhos. Na linha 29, o algoritmo recebe as informações enviadas pelo nó filho. Na linha 30, é verificado se o MBR da entrada que aponta para o nó filho é igual ao MBR enviado pelo nó filho.

As informações dos nós filhos são armazenadas em uma memória compartilhada, com controle de concorrência, pelas réplicas de T. Por isso, na linha 31, estas informações são obtidas da memória compartilhada. Na linha 32, são adicionados os dados coletados nas linhas 29 e 30 na lista de informações. Na linha 33, é obtido na mémoria compartilhada das réplicas de T o número de nós filhos que enviaram informações de depuração. Este valor é armazenado na variável $count$ e este valor $count$ é decrementado para que as outras réplicas tome ciência.

Se todos os nós filhos estiverem enviados a resposta, então a variável count irá valer 0 e as linhas 35 a 39 serão processadas. Se T for raiz, então serão enviadas as informações para a aplicação cliente, caso contrário, estas informações serão enviadas para o nó pai de T. Se a variável $count$ for maior que 0, então as informações dos clientes são armazenadas na memória compartilhada.

Algoritmo 1: $RDebug(T)$

Entrada: T reference to root node of R-Tree $tree$
Saída: Debugging information about distributed R-Tree $tree$

```
1  S1 [Search subtrees]
2  if T is not leaf then
3      store the number of child entries in each replica server of T
4      for each entry E in T do
5          server ← choose one server, randomically, that store one replica of E
6          send msg to server to process the node child of E on step S1
7      end
8  else
9      verifiy the consistency of T in others replicas
10     Invoke step S2 [Aggregation]
11 end
12 S2 [Aggregation]
13 replica_consistency ⇐ verifiy the consistency of T in others replicas
14 node_consistency ⇐ verify the consistency of M and m values of T
15 add in informations: replica_consistency and node_consistency
16 overlap ⇐ overlap area of T
17 dead_area ⇐ dead area of T
18 bound ⇐ MBR of T
19 list ⇐ ∅
20 for for each entry E in T do
21     add the MBR of E in list
22 end
23 if T is leaf then
24     if T is root then
25         send response with R-Tree nodes information to app client
26     end
27     send msg with informations to parent of T
28 else
29     entry_info ⇐ informations sent by child node
30     mbr_consistent ⇐ verify if the bound of the child node is equal to
       bound of entry of T that points to this child
31     informations ⇐ the child informations stored on shared memory by
       replicas of T
32     add in informations: entries_info, and mbr_consistent
33     count ⇐ retrieve the number of entries child which not sent a debugging
       response and decrement by 1 unit
34     if count == 0 then
35         if T is root then
36             send response with informations to app client
37         else
38             send msg with informations to parent of T
39         end
40     else
41         store informations on shared memory
42     end
43 end
```

O algoritmo 1 foi implementado na plataforma DistGeo para coletar as informações de depuração da R-Tree distribuída construída. Estas informações são utilizadas na plataforma para encontrar problemas de indexação e para otimizar o índice R-Tree para a execução de consultas. Uma aplicação gráfica, apresentada na Figura 4.1, foi construída para visualizar a estrutura do índice distribuído R-Tree construído, a partir das informações geradas pelo algoritmo de depuração distribuído construído na plataforma DistGeo.

[Colocar a Figura aqui!!]

Utilizando como base o algoritmo RDebug 1 é possível realizar a depuração dos algoritmos de busca sobre uma R-Tree. Por exemplo, o algoritmo de Window Query apresentado na Seção 2.1. Para adaptar o RDebug para a Window Query, basta acrescentar uma janela de consulta na primeira etapa e coletar as informações de agregação dos nós acessados na primeira etapa. Para depuração dos algoritmos que acessam diversas R-Trees, tais como o Spatial Join, o algoritmo deve ser modificado de forma drástica, pois o algoritmo pode acessar diversos caminhos diferentes.

4. Related works

De forma geral, nenhum trabalho foi realizado na proposta de técnicas de depuração do índice R-Tree distribuído, de acordo com nosso conhecimento e observando as revisões de literatura feitas em [Manolopoulos et al. 2003, Jacox and Samet 2007]. Nosso trabalho propõe um novo algoritmo de depuração do índice R-Tree distribuído e uma plataforma com arquitetura peer-to-peer para processar os algoritmos espaciais distribuídos.

Os trabalhos encontrados na literatura ou propõem técnicas de depuração distribuídas em aplicações genéricas ou propõem técnicas de processamento distribuído da R-Tree. Na Seção 4.1 são apresentados os trabalhos que propõem técnicas de depuração distribuída e em 4.2 são apresentados as propostas de plataformas para processamento distribuído de algoritmos espaciais.

4.1. Distributed Debugging 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.

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.

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 people 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.

4.2. Distributed Spatial Algorithms

Esta Seção apresenta os trabalhos que propõe algoritmo para processar os algoritmos espaciais de forma distribuída. A primeira publicação encontrada a respeito do tema foi a M-RTree [Koudas et al. 1996], que apresenta um servidor master e vários servidores escravos para processar as requisições. Isto gera uma grande concentração de processamento no servidor mestre, que além de processar alguns níveis diretórios da R-Tree, agrega as respostas para as aplicações clientes. Estes comportamentos também são evidenciados na MC-RTree [Schnitzer and Leutenegger 1999].

Em [An et al. 1999], os autores exploram a distribuição de uma árvore R-Tree em um cluster. O artigo propõe uma arquitetura similar à M-RTree e MC-RTree e, por isso, possui os mesmos problemas descritos anteriormente.

Na SD-RTree proposta em [du Mouza et al. 2007] uma árvore binária é utilizada ao invés de uma R-Tree. A árvore binária causa um grande aumento no número de mensagens, já que para o mesmo conjunto de dados a árvore binária possui um número bem

maior de níveis que a R-Tree. A proposta Hadoop-GIS [Kerr 2009] apresenta uma alternativa para processamento paralelo de dados espaciais, mas não emprega índices para melhorar o desempenho das operações nos datasets.

Em [de Oliveira et al. 2011] é proposta uma plataforma de processamento distribuído de operações espaciais. A arquitetura proposta não é escalável, já que todas as mensagens devem passar por um servidor master replicado. Em [saviosbrc] é proposta uma plataforma com arquitetura peer-to-peer híbrida para processar a junção espacial distribuída. A arquitetura proposta em [saviosbrc] possui um conjunto de servidores centrais para resolução de nomes e gerenciamento de eventos. Estes servidores podem se tornar um gargalo no sistema.

Um framework é proposto em [Xie et al. 2008] para realizar o processamento de consultas espaciais com um esquema de balanceamento de carga em duas fases. Em [Zhang et al. 2009] é proposta uma arquitetura que utiliza o modelo MapReduce para processar as operações espaciais. A arquitetura é indicada, segundo o próprio trabalho, em casos onde as bases de dados não estão indexadas. Uma arquitetura utilizando o modelo MapReduce é apresentada em [Zhong et al. 2012]. A plataforma possui um índice espacial distribuído em dois níveis: índice global e índice local.

Diferentemente da plataforma DistGeo proposta neste trabalho, nenhum trabalho encontrado na literatura implementou uma plataforma com arquitetura peer-to-peer para processamento de algoritmos espaciais distribuídos. Além disso, nenhum destes trabalhos propõem técnicas para depuração do índice espacial R-Tree de forma distribuída.

5. Conclusion

A plataforma DistGeo apresenta uma solução completa para o processamento distribuído de operações espaciais utilizando o índice R-Tree distribuído. Este processamento distribuído gera um desafio: como depurar o algoritmo de inserção em uma R-Tree com os nós distribuídos em um cluster de computadores.

Nenhuma técnica de depuração dos algoritmos espaciais da R-Tree foi encontrado na literatura. Por isso, neste trabalho foi proposta uma nova técnica de depuração da construção do índice R-Tree distribuído denominado RDebug. Esta técnica utiliza o próprio índice R-Tree para coletar as informações de depuração. Esta coleta é realizada no índice R-Tree realizando o caminhamento na árvore de forma down-top. Utilizando a própria estrutura distribuída do índice, os dados podem ser coletados de forma distribuída, aumentando a eficiência no processamento do algoritmo de depuração.

Esta nova técnica foi implementada na plataforma DistGeo que possui arquitetura peer-to-peer. Os nós da R-Tree estão distribuídos e replicados pelo cluster de computadores. Por isso, o algoritmo RDebug pode ser processado sem que haja pontos de gargalo e pontos de falhas no cluster. Além disso, a replicação dos nós da R-Tree no cluster permitem que seja realizado um balanceamento de carga no caminhamento do índice distribuído R-Tree. Neste caminhamento, em cada acesso a um nó da R-Tree pode-se escolher a máquina com menor carga de processamento naquele momento, aumentando a eficiência do algoritmo RDebug. Estas informações do estado da máquina são trocadas utilizando o algoritmo Gossip.

Uma aplicação gráfica foi implementada para visualizar a estrutura do índice dis-

tribuído R-Tree e as informações de depuração da construção do índice. Com estas informações é possível identificar inconsistência na construção do índice e otimizar a qualidade do índice distribuído.

Em trabalhos futuros, o algoritmo RDebug será modificado para depurar de forma distribuída os algoritmos de busca Window Query e Join Query. O algoritmo RDebug pode ser adaptado facilmente para coletar informações de depuração da Window Query. Para o algoritmo Join Query ele deve sofrer diversas mudanças, já que o caminhamento é realizado em duas R-Trees distribuídas diferentes. Serão realizados testes de desempenho para analisar a escalabilidade do algoritmo RDebug na plataforma DistGeo.

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.

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

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.

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.