Distributed Spatial Data Structuress

by

Zahra Miri Kharaji and Bradford G. Nickerson

Technical Report TR14-231

August 12, 2014

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
http://www.cs.unb.ca

Contents

Chapter 1

Introduction

1.1 Range Searching Problem

One of the fundamental areas of computational geometry is the range search problem. Range search problem is convincingly motivated by practical applications like GIS (Geographical Information Systems), CAD (Computer Aided Design) and databases. Before presenting a formal definition of range search problem, we need to speak about range and its various types.

Let R^d be d-dimensional Euclidean space and $\mathbf{R}$ be a group of subsets of R^d. Each member of $\mathbf{R}$ is called a range. Describing typical kinds of ranges may help for deeper understanding.
1. Orthogonal ranges (R_{orthog}) are axis parallel boxes which all sets are in the form of $\prod_{i=1}^{d}[a_i, b_i]$ where $a_1, b_1, ..., a_d, b_d \in \mathbb{R}$.
2. Half space range (R_{half}) is the set of all halfspaces in R^d space.
3. Simplex range is the set of all simplices in R^d where simplices is a generalization of triangle in d-dimensional space.
4. Ball range (R_{ball}) is the set of all balls in R^d [21].
Figure 1.1 shows examples of $\mathbf{R}$ in 2-dimensional space.

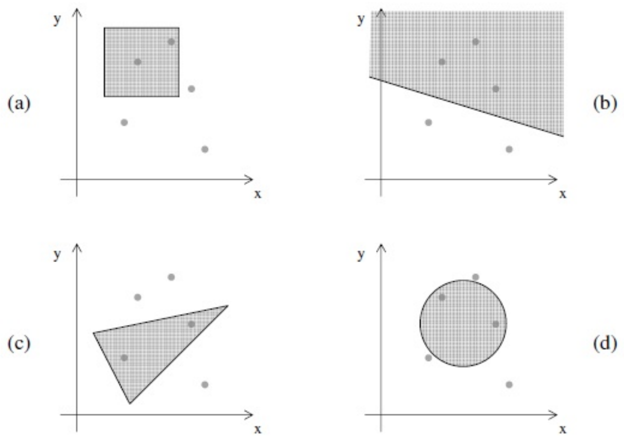


Figure 1.1: Examples of range search in 2-dimensions;(a) rectangular, (b) half-space, (c) simplex, (d) ball.

Figure 1.1(a) shows a specific example of rectangular ranges. To define a general form, let $R(d,k)$ be a rectangular range in d-dimensional space with k dimensions having finite interval where $0 \leq k \leq d$ [2]. According to this definition, Figure 1.1(a) shows a rectangular range $R(2,2)$. Figure 1.2 shows various rectangular ranges in 2 and 3 dimensional space.

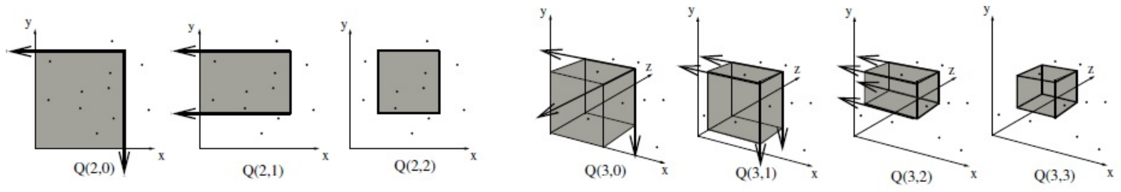


Figure 1.2: 2 and 3 dimensional rectangular range(from [2])

Let P be a set of N points in R^d and r is a range $\in \mathbf{R}$. The range searching problem is to design an efficient algorithm which reports all points of P lying in r.

Assuming a RAM model, the set of all points, P and the range r are available in main memory at the same time. The simplest range search algorithm goes through all points of P one by one and report points in the range. This "linear search" algorithm uses $\Theta(N)$ space and $\Theta(N)$ time. Typically, however, the point set P is given in advance and we want to answer different queries over the same set P several times. In range search problems, we typically want to preprocess N points into data structure in order to answer range queries as fast as possible.

Reporting points of the set P which intersecting with range r (range-reporting query) is just one of the possible range search problems. Counting points lying in a given range is called a range-counting query is another one. Sometimes we just want to check the emptiness of $P \cap r$ (range emptiness query). Another case of range searching problems is optimization range query. In this case, we are looking for a specific point with certain property among range's points [24].

In reality, spatial data objects usually occupy areas in multi-dimensional space. In 2-dimensional space, these objects are represented by points, triangles or polygons. In range searching, when objects are represented by triangles or polygons, we want to determine all the objects intersecting a given range. For example, on a map, objects like counties cover non-zero size regions in 2-dimensional space, we want to find all counties within 20 km of our location. Different representations of objects use various approaches for solving range search problems. in this work, we assume that all objects are represented by points.

1.2 Motivation and Applications

Different types of range-search problems are motivated from different practical applications. In the following we speak about a simple example which is presented in most of range search references for explaining the application of range search. Consider a Company employees data table. In this table, each record is related to one employee and can be interpreted as a point in d-dimensional space where each dimension corresponds to one field of the table. For example if we want to report all employee born between 1950 and 1955 who earn between \$3,000 and \$4,000 per month and have 2, 3 or 4 children, we could use a 3-dimensional space where each point shows an employee. Figure 1.3 shows this case. We can see that one coordinate is devoted to date of birth, the second one to salary and the last one to number of children. All the points lying in grey parallel box are the answer of an orthogonal query [20].

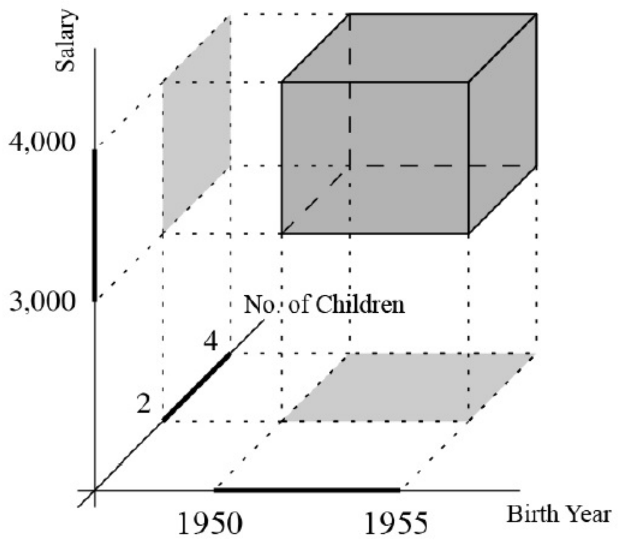


Figure 1.3: Orthogonal range in 3-dimensional space (from [20])

Locating an airport by a defective airplane in emergency condition can be an application of ball range search. In ball range search, all the ranges are in form of ball range and we are looking for all points within distance r from one specific point that is moving. More important than direct applications, range search has applications as subroutines in other geometric algorithms. For example, in facility location problem, we are looking for optimal placement of facilities in an area to minimize some distances and costs like transportation cost. Range searching algorithms may be used as subroutines in optimal placement of facilities algorithms.

Various works on computational geometry, generalize the range searching problem in order to merge all kinds of range searching and making a unified problem. Given P a set of points, we assign a weight $w(p) \in S$ to each point $p \in P$. If $(S, +)$ is a commutative semigroup, for any subset $P' \subseteq P$, $w(P') = \Sigma_{p \in \mathbb{P}'} w(p)$. For each query R, our purpose is computing $w(P \cap R) = \Sigma_{p \in \mathbb{P} \cap \mathbb{R}} w(p)$ by this definition all types of range searching problem can be defined. For example, for range counting query, S is the set of integers Z and $+$ is the standard integer addition. By getting $w(p) = 1$ for each $p \in P$, this problem has been defined. For emptiness query, semigroup is $(\{0, 1\}, or)$ and $w(p) = 1$ for each $p \in P$ [3].

The performance of a data structure is affected by three elements with different levels of importance. The first one is preprocessing time, P, the time that we need for constructing a data structure. Preprocessing time, P, is less important than the search cost (Q) and storage space (S).

The size of the data structure includes the original data size N plus auxiliary information required to create the data structure. In the RAM model, search cost is the time spent to answer a query and depends on number of points, N and d, the number of dimensions. In the I/O efficient model, search cost is measured by the number of disk blocks accessed during the search process and depends on N, d and B where B is the number of points in one disk block. For range reporting, query cost also depends on the number of reported points, K, in addition to other parameters. It should be mentioned that for dynamic data structures, the cost to update the data structure (insertion and deletion) may be important [13].

1.3 Objective

In this report, we try to do a comprehensive survey of recent developments on dynamic spatial data structures supporting orthogonal range search on a distributed computing model. We are looking for various constraints and advantages of different data structures and collecting some open problems in this area. In addition to the problem size N and number of dimensions d, the distributed computing model also considers the number of nodes n participating in the computation. Our focus is on data structures supporting d-dimensional points and robust range search that still provide correct results under failure of one or more nodes.

Chapter 2

Distributed Computing Model

In a formal definition, a distributed system is a collection of independent computer systems connected by a communication network which cooperate to achieve a common purpose (see e.g. [14]). Distributed computing is a model of computation which uses distributed systems to solve a computational problem. Special characteristics of distributed computing make it distinctive from other models, as follow [14]:
1. There is not any global clock in the system. It mean that all processors work asynchronously.
2. Each processor has its local memory and there is no shared memory in the system. Because of this important feature, message passing is required for communication.
3. Processors could be geographically separated from each other or be part of a cluster of workstations on a LAN.
4. Processors are autonomous from each other. Each of them can have different speeds and different operating systems.

In this study, due to the above properties of distributed systems, shared memory models are not considered.

Distributed systems are composed of n processing nodes, each of which has a its own memory and processing unit. Processing units can be a single processor or a shared memory multiprocessor. Figure 2.1 shows a typical distributed system. Processing nodes are connected by a communication network.

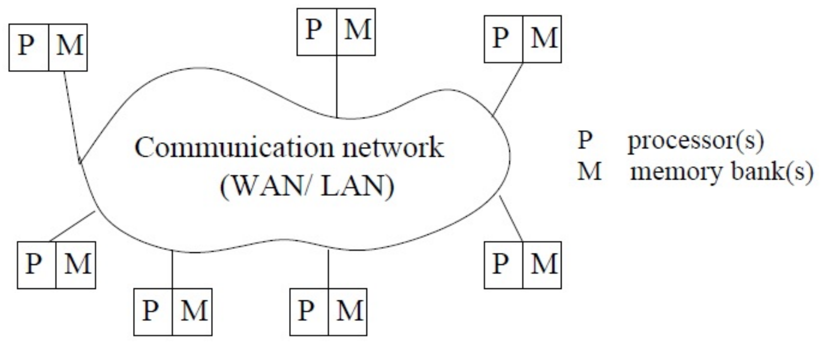


Figure 2.1: A typical distributed system (from [14]).

Use of distributed systems can enhance reliability significantly. It's clear that the possibility of malfunction of distributed resources at the same time is low. In addition, replicating resources for increasing reliability is common. Availability, fault-tolerance and integrity are other advantages of a distributed model which follow reliability. In some cases, data cannot be replicated at all nodes of a distributed system because of its size or sensitivity. Distributed systems provide the conditions for accessing geographically remote data and resources. Furthermore, distributed systems benefit from scalability which means increasing the number of processors without losing efficiency.

A distributed program consists of p asynchronous processes which are separated among n processing units in a distributed system. If the number of processes, p, is greater than number of processors, n, more than one process will be devoted to one processor. Each process uses two kinds of channels for communication with other processes. Processes running on the same processor use *internal channels* to exchange messages among themselves and processes running on different processors use *external channels* for communication. For running a process, a processor executes its different actions sequentially. In [12] various actions of a process are divided into three events; internal events, message send events and message receive events. Figure 2.2 shows the progress of three processes p_1, p_2 and p_3 over time. In this figure e_i^x is a representation of xth event at processor p_i. Horizontal lines and dots represent time and events respectively. An arrow indicates message transfer and three types of events are described. The second event of process p_1 is a message send event, the third event is an internal event, and the fourth event is a message receive event. Data transferred in distributed systems are copied to another node using message send and message receive events.

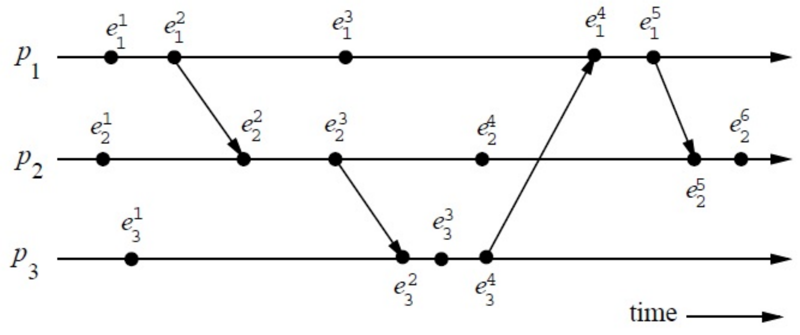


Figure 2.2: The space-time diagram of a distributed execution. (from [14])

2.1 Routing in Message Passing network

The distributed model is based on communication of processing nodes by message passing over an interconnection network. The performance of this system is related to how messages get to their destinations. Routing is a mechanism for determining a path among all the eligible paths for a message to get to a destination node from a source node. The routing mechanism gets the addresses of source and destination nodes as input. In addition, depending on the type of routing mechanism, it may require current sate of network as another input. The output of the routing mechanism is one or more paths over the network from source to destination.

Routing mechanisms can be categorized based on different criteria. Based on the length of chosen path, routing mechanisms are classified as *minimal* and *non-minimal*. A minimal routing mechanism returns one of the shortest paths from source node to destination node and may cause congestion in part of the network. A non-minimal mechanism considers the current state of the network and may return a longer path to avoiding network congestion.

Routing mechanisms can also be classified based on their input into two groups; *Adaptive* and *Deterministic*. Deterministic routing mechanisms use only source and destination node addresses to determine the path. They don't use any information about the current state of network. In comparison, adaptive routing mechanisms depend on network conditions to determine a path between source and destination nodes for message passing.

In addition to previous classifications, based on the method used to make routing decisions, routing techniques can be categorized as *centrilized* and *distributed* groups. In the centralized approach, the entire path for a message is determined at the beginning of the path. To use this technique, each processing node needs to know about the status of the other nodes. In contrast, in a distributed approach, each node determines which channel should be used for forwarding messages. So, each processing node just needs knowledge about the state of neighbouring nodes.

One of the deterministic minimal routing mechanism is *dimesion order* routing. In this method, messages traverse a network, dimension by dimension. For a 2-dimensional mesh, dimension order routing which is called $XY-routing$, first send message along X dimension and then along Y dimension. For example, assuming $[S_x, S_y]$ and $[D_x, D_y]$ are the coordination of source and destination nodes respectively. The message is passed along X dimension until it reaches $[D_x, S_y]$ and then along the Y dimension until it reaches the destination. Figure 2.3 shows a sample of XY-routing [15].

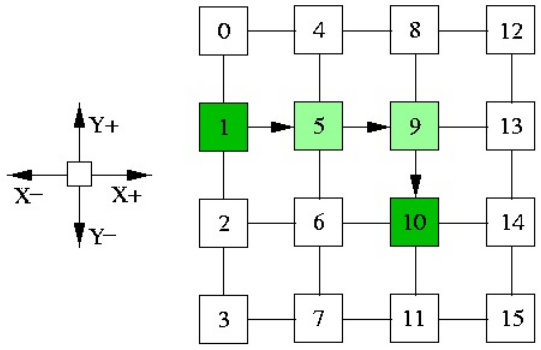


Figure 2.3: XY-routing in 4 × 4 mesh.

2.2 Switching Mechanisms

In distributed systems, switching mechanisms as a factor affecting network performance have been considered. Switching mechanisms are defined as techniques for transforming data from an input channel into output channel. Various switching mechanisms with different drawbacks and advantages are used. In the following, some common techniques are discussed.

Circuitswitching is one of common techniques which reserve a path between source and destination processing nodes before sending messages, and release reserved links after message exchange. This technique allocates static bandwidth for transferring messages over a communication network. Such static bandwidth allocation, regardless of real need, decreases performance in this model. Circuit switching techniques benefit from a simple buffering strategy and guarantees maximum latency. Latency is an important parameter in communication networks. It measures the time to complete a message transfer through the network. In addition, among all switching mechanisms, circuit switching has the smallest amount of delay in message routing.

Another alternative switching scheme is called $store - and - forward$. A major characteristic of store-and-forward methods is dynamic bandwidth allocation. Two common types of store-and-forward networks are $packet - switched$ and $virtual\ cut - through$. In packet-switched networks, before sending message, each message is broken into limited size packets and then packets are sent over an interconnection network. Routing each packet independently may cause different packets to follow different paths over the network and arrive at their destination out of order. So we need more overhead for each packet, in order to reassemble packets at the destination processing node. In the packet-switched technique, because of dynamic bandwidth allocation, all links are not available at the start of routing. Each node needs enough buffers to hold packets until the next link becomes available.

Virtual cut-through is another kind of store-and-forward mechanism that tries to improve the draw-backs of a packet-switched scheme. In the virtual cut-through method, each intermediate node stores the packet if the next selected link is busy; otherwise the packet is forwarded to the next node before the whole packet has been received. This method reduces latency and the required buffer size in each node. In virtual cut-through switching, a message is divided into fixed size parts smaller than packets in packet-switching called flits (flow control digits). The first flit called the header flit carries information about the destination node and determines the path for all subsequent flits. In his way, all the flit will be delivered in order. In addition, if the header is blocked, other flits will also be blocked. This is another reason why nodes in the virtual cut-through technique have less buffer space in nodes.

2.3 Message Passing Programming Model

In message passing programs, processes communicate with others by *send* and *receive* operations. There are different programming languages and libraries to implement send and receive operations and message passing protocols. Data exchanging varies from one program to another. At first glance, it may seem simple to ensure safe data transfer. For example, if process p_0 sends a message containing the value of A to process p_1 and after send operation, process p_0 changes the value of A immediately, the program should ensure that the value received by p_1 is equal to the value of A at the time of the send operation. In this section we will discuss different message passing communication protocols, their drawbacks and advantages.

2.3.1 Blocking Message Passing Operations

Blocking send operations until the program makes sure that the next operation is safe to do, is one simple way to avoid violation in previous example. Based on this simple idea, there are two approaches for implementing send/receive operations, *blocking non-buffered send/receive* and *blocking buffered send/receive.*

In the blocking non-buffered send/receive method, the send operation doesn't return until the receive operation is executed on the receiver process and the data will transfer from source node to destination node. This method usually follows this scenario; sending process asks receiver to send a message, when the receiving process replies to the sender's request, then the sending process will send data. Figure 2.4 shows handshaking in blocking non-buffered send/receive. In this figure, three different states are shown. In (a) and (c), sender and receiver begin communication at different times. We see that in these cases one of the processes is idle for a period of time. This idle state is the main drawback of this technique.

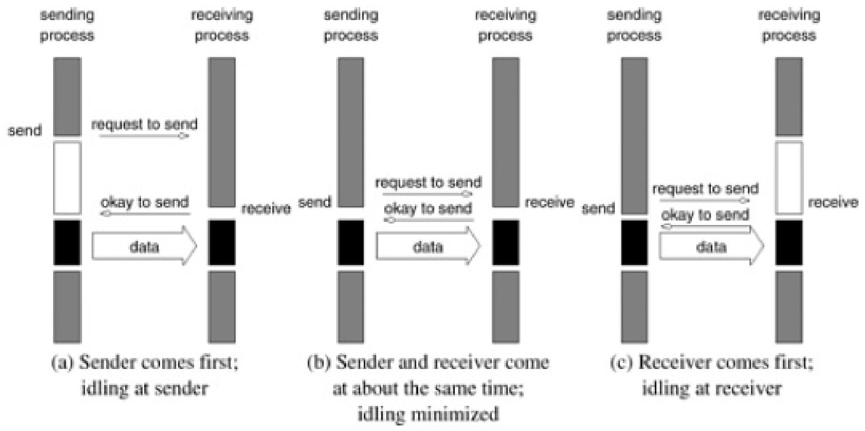

sending process
receiving process
send
request to send
okay to send
data
receive
(a) Sender comes first; idling at sender
(b) Sender and receiver come at about the same time; idling minimized
(c) Receiver comes first; idling at receiver

Figure 2.4: Handshaking for a blocking non-buffered send/receive operation (from [15]).

The second way for blocking message passing is blocking buffered send/receive. In this method the main drawback of first method is resolved by buffering. The sender and receiver have specific buffers allocated for communicating messages. First, the sender copies the data into a pre-allocated buffer. The sending operation is blocked until copying is completely done. After this step actual communication will be done from source to destination using the interconnection network. In this step data is stored in a buffer at the receiver end. When the receiver process encounters receive operations, it checks its own buffer and copies data into the target location. In comparison with non-buffered send/receive, here we are solving the idling time problem at the cost of adding buffer and buffer management. Figure 2.5

shows the blocking buffered transfer protocol assuming the existence of buffers at send and receive ends.

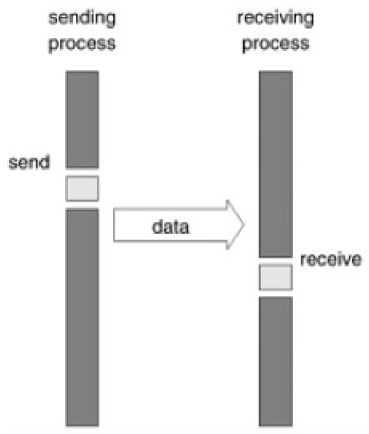


Figure 2.5: Blocking buffered transfer protocol (from [15]).

2.3.2 Non-Blocking Message Passing Operations

In blocking message passing implementations, send and receive operations return when it is safe to do so. In other words, further changes on transmitted data don't violate program. In non-blocking message passing operations, it is the task of the programmer to make sure the program is correct. In this case, send or receive operations return before it is safe to so. The programmer should not change sent or received data before completion of the transfer. The state of send and receive operations can be determined using check-status operations. This operation shows whether using specific data is permissible or not. Immediately after returning from send and receive operations, processes do any computation which is independent from send and receive.

Blocking and non-blocking operations are two common implementations in message passing. Both are implemented in typical message passing libraries like Message Passing Interface (MPI). Blocking operations are safer and easier in comparison with non-blocking protocols. Non-blocking operations are more appropriate for performance optimization.

Chapter 3

Rainbow Skip Graph

3.1 Introduction to Skip List and Skip Graph

3.1.1 Skip List

A *Skip List* is a data structure constructed from some linked lists connected to each other containing a subsequence of ordered sequences of elements. Each linked list skips some elements of previous linked lists and result in an increasingly sparse set of linked lists. A skip list uses probabilistic balancing, and elements to be slipped are chosen probabilistically.

Figure 3.1 shows the basic idea behind of skip list data structure. When we are searching for a specific element a linked list in storing N sorted elements, every node has to be explored in worst case scenario (Figure 3.1.a). If every second node stores a pointer to the node two ahead, $\lceil \frac{n}{2} \rceil + 1$ nodes have to be explored in the worst case (Figure 3.1.b). To generalize this rule, consider that every 2^ith node stores a pointer to the 2^i node ahead, we need to explore $\lceil \log_2 n \rceil$ nodes in the worst case. This data structure works well for searching but insertion and deletion are impractical.

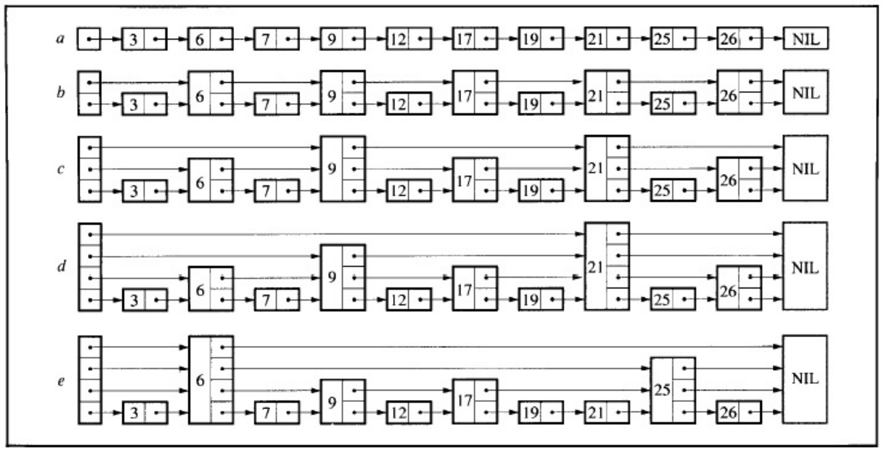


Figure 3.1: Linked list with additional pointers (from [25]).

In the above data structure if we randomly choose the nodes which contain a pointer to some node ahead, we achieve a probabilistically balanced data structure called the skip list. Figure 3.2 shows an example of a skip list. The bottom linked list contains all sorted elements. For all higher layers, an element in layer i is in layer $i+1$ with probability p. Two commonly chosen values for p are $\frac{1}{2}$ or $\frac{1}{3}$. The first layer of a skip list for n elements stores n pointers, the next layer stores np pointers, and the next one stores np^2 pointers, etc. Using a geometric series equation, the number of stored pointers required for a skip list is $\frac{n}{1-p}$, on average. If $p = 1/2$, we require 2 pointers per element on average. The space required for a skip list $S(N) = O(N)$ with high probability.

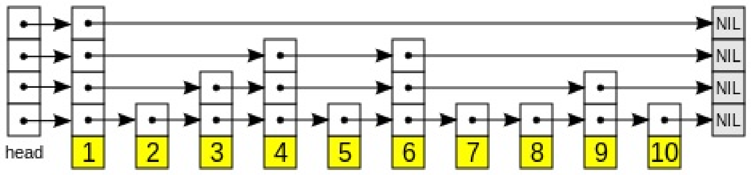
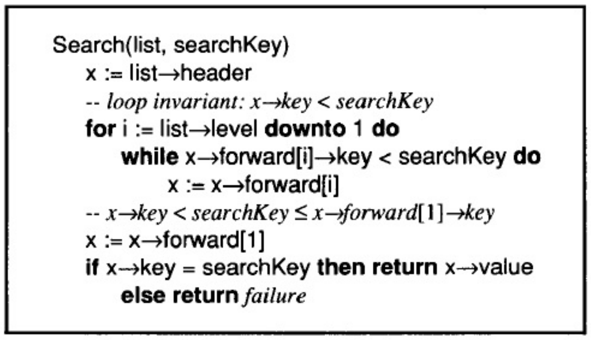


Figure 3.2: An example of skip list

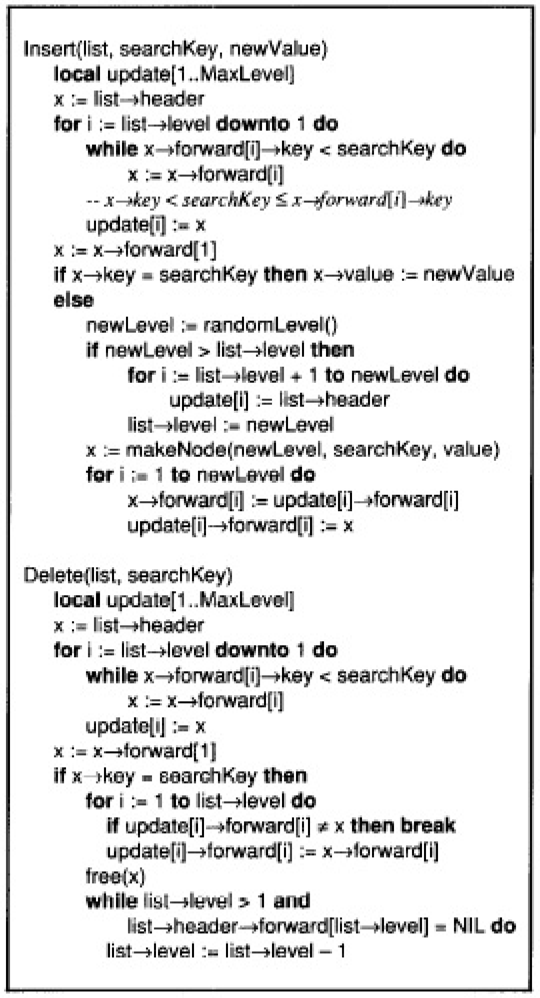
Figure 3.3 shows the search algorithm for a skip list. This algorithm starts from the tallest level of the skip list and traverses this level horizontally until no more progress can be achieved. In other words, when all the remaining nodes in the current level has a key greater than the search key, the search algorithm moves down to the next level and does the same operation in current level until it reaches level 1. In level 1, when no more progress can be made, we find the desired key and return the stored value, or the desired key isn't in the list and return failure.



```
Search(list, searchKey)
    x := list→header
    -- loop invariant: x→key < searchKey
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
    -- x→key < searchKey ≤ x→forward[1]→key
    x := x→forward[1]
    if x→key = searchKey then return x→value
        else return failure
```

Figure 3.3: Skip list search algorithm (from [25]).

To insert a value in a skip list, we need to find the key and update the value. In the case when the key is not found, we need to add a new node in the proper place. The level of node is randomly chosen. Figure 3.4 shows insertion and deletion algorithms for a skip list.



```
Insert(list, searchKey, newValue)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        -- x→key < searchKey ≤ x→forward[i]→key
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then x→value := newValue
    else
        newLevel := randomLevel()
        if newLevel > list→level then
            for i := list→level + 1 to newLevel do
                update[i] := list→header
            list→level := newLevel
        x := makeNode(newLevel, searchKey, value)
        for i := 1 to newLevel do
            x→forward[i] := update[i]→forward[i]
            update[i]→forward[i] := x

Delete(list, searchKey)
    local update[1..MaxLevel]
    x := list→header
    for i := list→level downto 1 do
        while x→forward[i]→key < searchKey do
            x := x→forward[i]
        update[i] := x
    x := x→forward[1]
    if x→key = searchKey then
        for i := 1 to list→level do
            if update[i]→forward[i] ≠ x then break
            update[i]→forward[i] := x→forward[i]
        free(x)
        while list→level > 1 and
                list→header→forward[list→level] = NIL do
            list→level := list→level – 1
```

Figure 3.4: Skip list insertion and deletion algorithms (from [25]).

Table 3.1 shows the time and space complexity of the skip list data structure. Although the skip list data structure has a very bad worst case complexity, the probability of having an unbalanced skip list that gives rise to the worst case scenario is very low. The term, "with high probability" is a technical term which is used in analysis of randomized algorithms. In a formal definition, an event E occurs with high probability, if it occurs with probability at least $1 - O(1/n^c)$ for some constant $c > 0$ and independent from n.

3.1.2 Skip Graph

Skip graphs are a kind of distributed data structure which use the skip list idea. In comparison to skip lists where each level contains one linked list, in skip graphs each level contains multiple linked lists. In addition, in a skip list each level skips over some elements of the

Table 3.1: Skip list space and time complexity (from [25]).

	With High Probability	Worst case
Space	$O(n)$	$O(n \log n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

previous level but in skip graphs all nodes participate in every level. A skip graph has $O(\log n)$ levels, on average and the highest level consists of singletones, i.e. linked list with one element. Figure 3.5 shows an example of a skip graph.

In a skip graph, different nodes can be stored at geographically separate locations and it supports failure of nodes at any time. Skip lists don't work well in distributed systems. as they suffers from lack of redundancy. In the skip list, since each node is connected to only $O(1)$ other nodes and just a few nodes appear in the higher levels, failure of nodes affects the properties of data structure significantly and makes some nodes isolated. Skip graphs generalize the skip list data structure to define another data structure supporting distributed nodes and unpredictable failures.

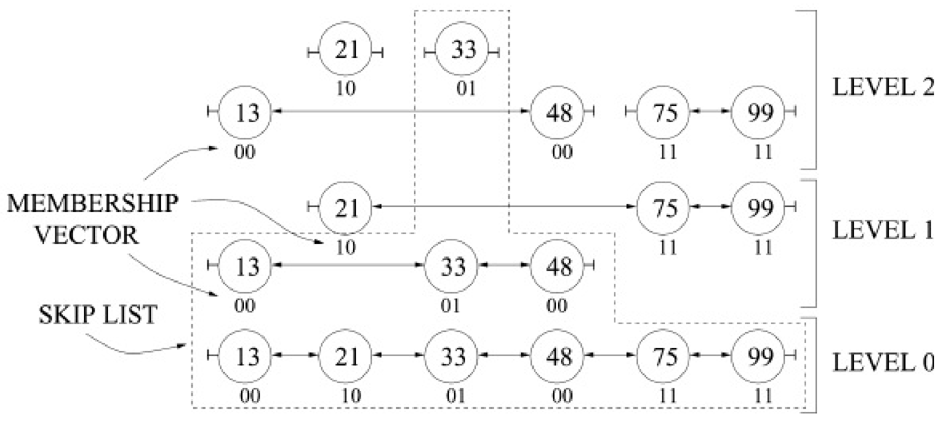


Figure 3.5: An example of skip graph (from [4]).

A skip graph is like a collection of skip lists that share some lower levels. Searching a skip graph uses the lists containing the starting element of the search. The set of all lists which contains a specific elements in a skip graph constructs a skip list. For example, in Figure 3.5 if we are looking for key 75 starting from node 33, the search path is restricted to the skip list in this figure. That is the reason why the search, insertion and deletion algorithms of skip graphs are similar to the algorithms of skip lists and require logarithmic time. In addition, each node in a skip graph needs logarithmic space to store pointers.

Each of the nodes in a skip graph has a membership vector which controls members of lists in the skip graph's levels. Usually, the membership vector is a binary sequence of $O(\log N)$ length. At level i of the skip graph, each node is connected to other nodes that have the same prefix of length i in their membership vectors.

One of the most important properties of a skip graph is fault tolerance. Experimental results in [4] shows that the skip graph is highly resilient against failure of nodes. In this experiment, in a skip graph of $131,072$ nodes, nodes are chosen to fail in a random pattern Figure 3.6 shows the result of this experiment. We see that as the probability of node failure is increased (X axis), the size of primary component as well as the fraction of isolated nodes over total number of nodes is measured (Y axis). Primary component is the fraction of total nodes in the largest connected component of live nodes and isolated nodes is the fraction of total nodes in the graph not connected to the primary components. Figure 3.6 shows that the number of isolated nodes in a large skip graph is almost zero until two-third of the nodes fail.

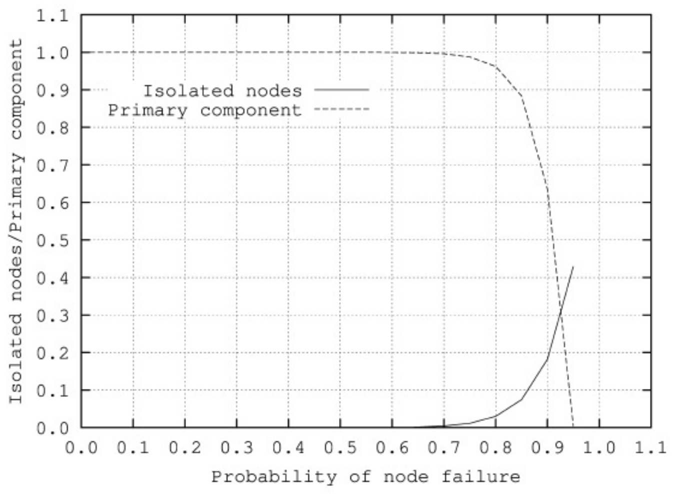


Figure 3.6: The number of isolated nodes in a skip graph in an experiment with 131,072 nodes (from [4]).

Table 3.2 shows the complexity of skip graph algorithms in term of time and number of messages.

Table 3.2: Skip graph complexity (from [4]).

	Expected time	Expected number of messages
Search	$O(\log n)$	$O(\log n)$
Insert	$O(\log n)$	$O(\log n)$
Delete	$O(\log n)$	$O(1)$

3.2 Non-redundant Rainbow Skip Graph

A non-redundant rainbow skip graph partitions n nodes into $\Theta(n/\log n)$ supernodes according to their keys, assigns a specific key to each supernode and constructs a skip graph by considering supernodes as a node in a skip graph. Each supernode S consists of $\Theta(\log n)$ nodes in a doubly-linked list which is called a *core list*. The smallest key of ordered keys in a core list is the key of a supernode. Each node of a supernode S participates in one level of a skip graph as a representative of the supernode in that level. S_i is the element of supernode S in level i and is connected to S_{i-1} and S_{i+1} to construct another linked list called *tower list*. A non-redundant rainbow skip graph is a skip graph constructed over supernodes. As explained in previous section, each level of skip graph contains different level lists. The same notion of level lists in a skip graph is used for a non-redundant rainbow skip graphs. Each node of a non-redundant rainbow skip graph participates in at most three lists, core list, tower list and level list.

Figure 3.7 shows an example of a non-redundant rainbow skip graph. Three different types of lists of a non-redundant rainbow skip graph are shown in this figure. Dashed lines shows different level lists in different levels and the sets of contiguous squares in level 0 refer to core lists. Arcs which are called rainbow connections shows the connections between nodes in a core list and their copies in a related tower list.

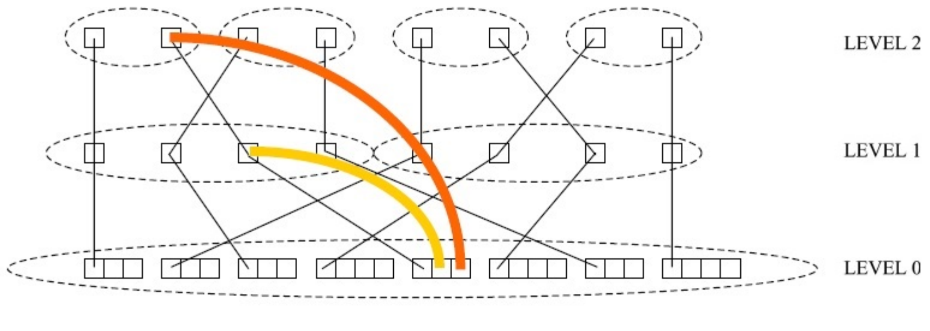


Figure 3.7: An example of rainbow skip graph (from [8]).

To search for a specific key in a rainbow skip graph, the query node sends the key to the topmost node of its own tower list. Using the skip graph search algorithm, the predecessor of the search key in the set of supernode keys is found. The found key is the key of the supernode whose the search key belongs. By searching the list of corresponding supernodes, the search key (or the next largest key, if the search key is not found) is found and returned to the query node. The required time for searching a rainbow skip graph is $O(\log n)$ as long as the size of every supernode is $O(\log n)$.

Updating a non-redundant rainbow skip graph is more complicated than updating a skip graph or a skip list. The complication arises from keeping the size of supernodes in $O(\log n)$ after insertion and deletion of nodes. The technique presented in [8] for achieving this goal is using the merge/split method used in updating B-trees. In this method, two constants c_1 and c_2 are chosen, where $c_1 < c_2$ and the number of nodes in a supernode always must be between $c_1 \log n$ and $c_2 \log n$. We should check the size of a supernode after every deletion and insertion and merge a supernode with its adjacent supernode when the size of a supernode is less than $c_1 \log n$ and split a node into two supernodes when the size of supernode is more than $c_2 \log n$.

Another requirement of updating a non-redundant rainbow skip graph is estimation of the value $\log n$ at each supernode without sending message to every supernode after each insertion and deletion. It is proven [9] that the number of levels in a skip graph is $\log n$ with high probability. Ideally, the height of supernodes in a non-redundant rainbow skip graph can be used as an estimation of $\log n$ but updating a skip graph may change the height of nodes. For solving this problem, a specific range is considered for the height of supernodes and when the height of one of the supernodes is outside of this range, we recompute the number of nodes in the skip graph using $\Theta(n)$ messages and rebuild the entire graph.

The main improvement of a rainbow skip graph compared to a skip graph is reducing the size of the nodes. As explained above, each node of a skip graph is required to store $\log n$ pointers while a rainbow skip graph has constant size nodes.

3.3 Rainbow Skip Graph

Using rainbow connections in a non-redundant skip graph with hydra components and erasure codes as explained in this section, we augment the non-redundant skip graph to make a new data structure called the rainbow skip graph (RSG). The RSG support failure of nodes which would render the non-redundant rainbow skip graph unsearchable. In the node failure process, we assume that each node with a constant probability less than 1 may leave the graph without notifying its neighbours or providing information necessary for restoring the structure of the graph.

Hydra components consist of a group of nodes which are connected to each other in such a way that deleting a constant fraction of nodes in the components with high probability leaves large connected subcomponents with high probability. Using erasure codes stored in the subcomponent, remaining nodes refresh the overlay network (deleting failed nodes) and compute new links between remaining nodes. In the following, we first present a formal definition of hydra components and introduce erasure codes, and then explain how to partition the non-redundant skip graph into hydra components to construct a rainbow skip graph [9].

The structure of a hydra component is based on $2d$-regular graphs. A $2d$-regular graph is a graph whose the set of edges is composed of d Hamilton cycles. Figure 3.8 shows an example of a $2d$-regular graph with $d = 3$. The set of all $2d$-regular graphs with n vertices is $H_{n,d}$. A (μ, d, δ)-hyda component is a set of μ nodes connected to each other based on one random graph of $H_{\mu,d}$, where $\delta < 1$.

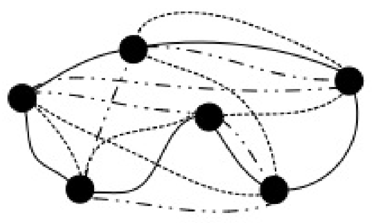


Figure 3.8: An example of a $2d$-graph consisting of 3 Hamilton cycles (from [16]).

An (n, c, l, r)-erasure-resilient code consists of two algorithm, encoding algorithm and decoding algorithm. The encoding algorithm gets an n-bit message as input and produces a set of l-bit packets with total length cn. The decoding algorithm recovers the original message from any set of packets with total length at least rn.

For a hydra component, we put the critical information (the information we need to restore the structure of a rainbow skip graph after failure of some nodes) in a message with M bits. Then, using (n, c, l, r)-erasure-resilient codes, the message is encoded into a new message with cM bits. The packets of the encoded message are equally distributed among μ nodes of the hydra component. The parameters (n, c, l, r) of the (n, c, l, r)-erasure-resilient code must be chosen in such a way that the original message can be restored from the information stored on any set of $\delta\mu$ nodes of each hydra component. Given the number of required nodes for restoring the original message after failure of up to $(1 - \delta)\mu$, $\delta\mu$, and the number of bits stored on each node, (i.e. cM/μ bits), we get that δ is equal to r/c.

To complete a non-redundant rainbow skip graph and achieve a rainbow skip graph, we need to explore how to partition the nodes of a non-redundant skip graph into hydra components. In a non-redundant skip graph we consider two kind of hydras, level-list hydras and supernode hydras. In [9], $9\beta\log(n')$ and $27\beta\log(n')$ are considered as minimum and maximum size of hydras respectively where $\beta = \mu/\log n$. For the level-list hydras, the elements of level lists based on their orders in the lists are placed in hydra components. In the case that the element of a list are not enough for constructing a hydra component, the elements in the beginning of the next list in the same level are added to hydra component. In the same way, the elements of a core list and tower lists construct the supernode hydras and when the size of hydra is too small, the nodes of adjacent supernod will be added to the hydra.

Chapter 4

Peer-To-Peer Indexing Structures

4.1 Peer-To-Peer Networks

Peer-to-peer(P2P) networks are a class of networks without any central point that uses distributed resources to achieve a specific purpose. Each node called a peer in the network is a consumer and supplier of data. As there is no single failure points in P2P systems, they are more reliable and scalable in comparison with client-server systems. Communication in P2P networks is done in a distributed manner and all the participants are peers. A peer-to-peer network is illustrated in Figure 4.1.

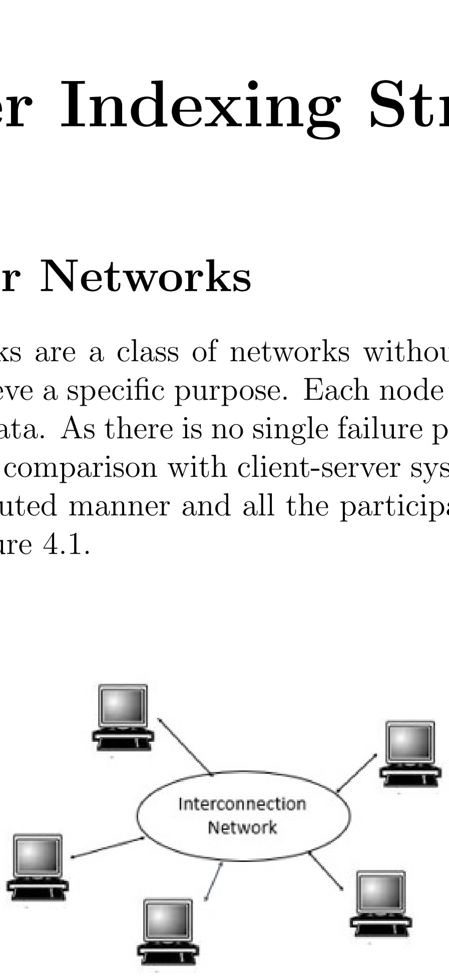


Figure 4.1: A peer-to-peer network.

4.2 Peer-To-Peer Indexing Data Structures

Recently there has been some effort toward development of distributed data structures for range search in multiple dimensions. In this chapter, we study two schemes supporting range query on peer-to-peer networks. The first data structure in [18] is a delay-bounded range search scheme which works over FissionE, a constant degree distributed hash table (DHT).

The presented scheme called *Armada* can support both single attribute and multi-attribute range queries. The second approach in [5] uses the non-redundant rainbow skip graph to implement an orthogonal search in 2-dimensional space.

4.2.1 Armada

Recently, some peer-to-peer systems use a DHT as general indexing scheme to support different applications. As a result, they inherit DHT's strong properties like scalability and efficiency. As DHTs support exact-match query, to process a range query we need a new scheme that works on top of the DHT. Armada is a delay bounded range search scheme that uses FissionE [19] as an underlying DHT to organize peers in an overlay network. The average degree of FissionE is 4 and its average path length is about $\log n$. In addition FissionE supports dynamic insertion and deletion of peers in less than $3 \log n$ messages where n is the number of peers.

FissionE supports an exact-match query. Armada on top of FissionE provides support for single attribute and multiple attribute range queries without modifying the structure of the underlying FissionE DHT. In this section, we first give an overview of FissionE and then we discuss the components of Armada.

The topology of FissionE is based on Kautz graphs. In this topology, the identifiers of peers and objects stored on peers are Kautz strings. The string $u_1u_2...u_k$ of length k and base d is a Kautz string where $u_i \in \{0, 1, 2, ..., d\}$ and $u_i \neq u_{i+1}$ $(1 \leq i \leq k-1)$. All Kautz strings of length k and base d create the $KautzSpace(d, k)$ of size $d^k + d^{k-1}$. The Kautz graph $K(d, k)$ is a directed graph whose nodes are labeled by strings in $KautzSpace(d, k)$. Each node $U = u_1u_2...u_k$ of a graph has an out-degree d to the nodes $V = u_2u_3...u_k\alpha$ where $\alpha \in \{0, 1, ..., d\}$ and $\alpha \neq u_k$. There is an outgoing edge from U to V iff V is a left-shifted version of U [19].

A Kautz graph is a static topology and needs some adjustments to be used for dynamic peer-to-peer networks. In FissionE, peers are organized in the form of an approximate Kautz graph using their Kautz strings as identifiers. The identifiers of peers in FissionE are Kautz strings of base 2. The length of identifiers may be different for different peers and the maximum length is less than $2 \log N$. Organization of nodes in FissionE is based on a defined rule called *neighborhood invariant*. Based on this rule, the difference of length of identifiers of every two neighbours must be one or less. Figure 4.2 shows an example of FissionE topology for 12 peers.

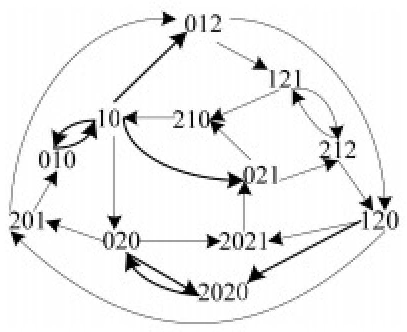


Figure 4.2: An example of FissionE topology (from [18]).

Armada uses the overlay network of FissionE and presents two parts. The first part uses *order-preserving naming* algorithms to assign object IDs from the Kautz namespace to the objects in order to distribute them on corresponding peers. The second part uses *range query processing* forward queries to the appropriate peers and return the results. In the following, we discuss order-preserving naming and range query processing algorithms for multiple attribute range queries.

4.2.1.1 Multiple_hash Algorithm

In order to support range query, naming algorithms that generate objectIDs for objects must keep the locality of attribute values. Order-preserving algorithms in Armada assign adjoining objectIDs in Kautz namespace to objects with close attribute values.

Before discussing the order-preserving naming algorithm, *Multiple_hash* for multiple attribute objects, we give some definitions. In addition, we assume that $\prec$ symbol provides the no more than relation between Kautz strings in lexicographical order. We assume that each object has m attributes, $A_0, A_1, ..., A_{m-1}$ and the values of objects are in m-dimensional subspace $w = <r_0, r_1, ..., r_i, ..., r_{m-1}>$ where r_i is the domain of values of attribute A_i.

Definition 4.1. The Kautz region $[\![\alpha, \beta]\!]$ is the subset of $KautzSpace(2, k)$ which includes all s that are in $KautzSpace(2, k)$ and $\alpha \prec s$ and $s \prec \beta$.

Definition 4.2. For two objects in m-dimensional subspace w, $\delta_1 = <u_0, u_1, ..., u_{m-1}>$ and $\delta_2 = <v_0, v_1, ..., v_{m-1}>$, $\delta_1 \lhd \delta_2$ if for each $0 \leq i < m - 1$, $u_i \leq v_i$.

Definition 4.3. Surjective function F from multiple dimensional space D to Kautz namespace V is multiple attribute partial order preserving iff for any δ_1 and δ_2 in D, if $\delta_1 \lhd \delta_2$ then $F(\delta_1) \prec F(\delta_2)$.

The partition tree presented in [18] tries to explain the *Multiple_hash* algorithm to assign partial order preserving objectIDs to objects. Partition tree $P(2,k)$ partitions the entire m-dimensional space w into smaller subspaces and maps the subspace in each leaf node to a specific Kautz string. Partition tree has $k+1$ levels and the root node represents the entire space w. The root node has three children while other intermediate nodes have two children. The label of edges in a partition tree depends on the label of the parent node. Edge label can be 0, 1 or 2, increasing from left to right. All nodes have a specific label at level j of the partition tree and the space covered by each node is divided into two subspaces along the ith attribute where $i = j \bmod m$ with m is the number of attributes for each object. Figure 4.3 illustrates an example of partition tree $P(2,4)$ for 2-dimensional space $<[0,6],[0,8]>$.

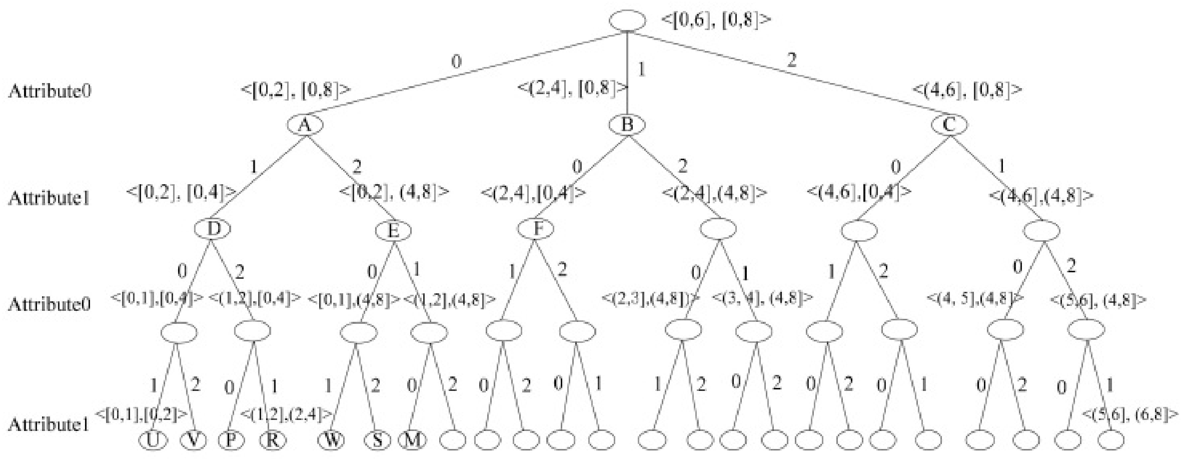
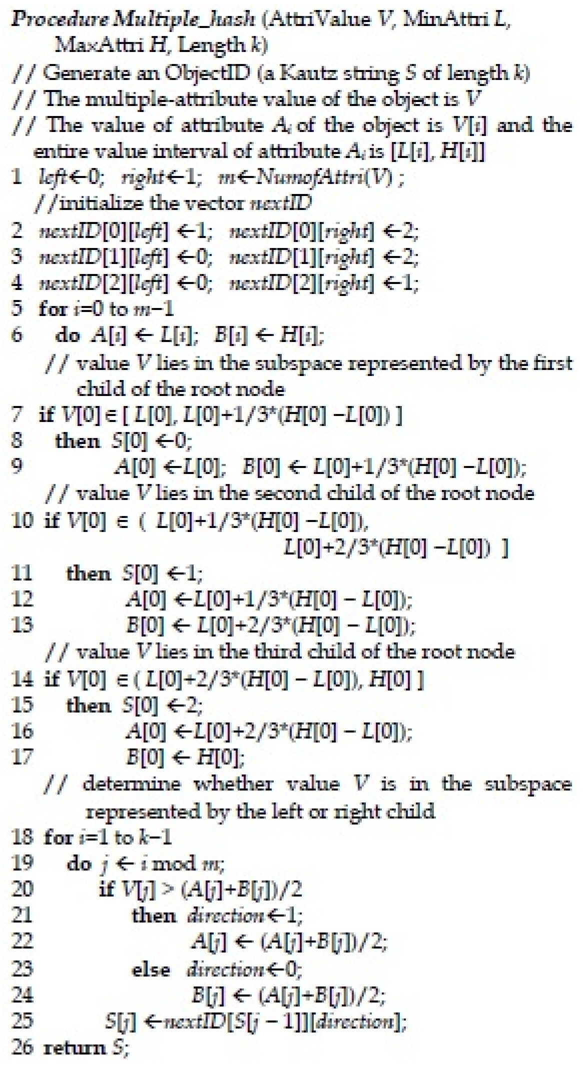


Figure 4.3: An example of partition tree $P(2,4)$ for 2-dimensional space $<[0,6],[0,8]>$(from [18]).

It is worth noting that in the *Multiple_hash* algorithm, we don't need to construct the partition tree. Partition tree is a model presented to design the *Multiple_hash* algorithm. As explained the *Multiple_hash* algorithm maps the multiple dimensional space to Kautz strings in $KautzSpace(2,k)$. So to publish objects in Armada, we first provide the ObjectID using *Multiple_hash* algorithm.Then we use the FissionE routing approach to find the unique peer whose peerID is the prefix of the objectID. The pseudocode of the *Multiple_hash* algorithm is shown in Figure 4.4.



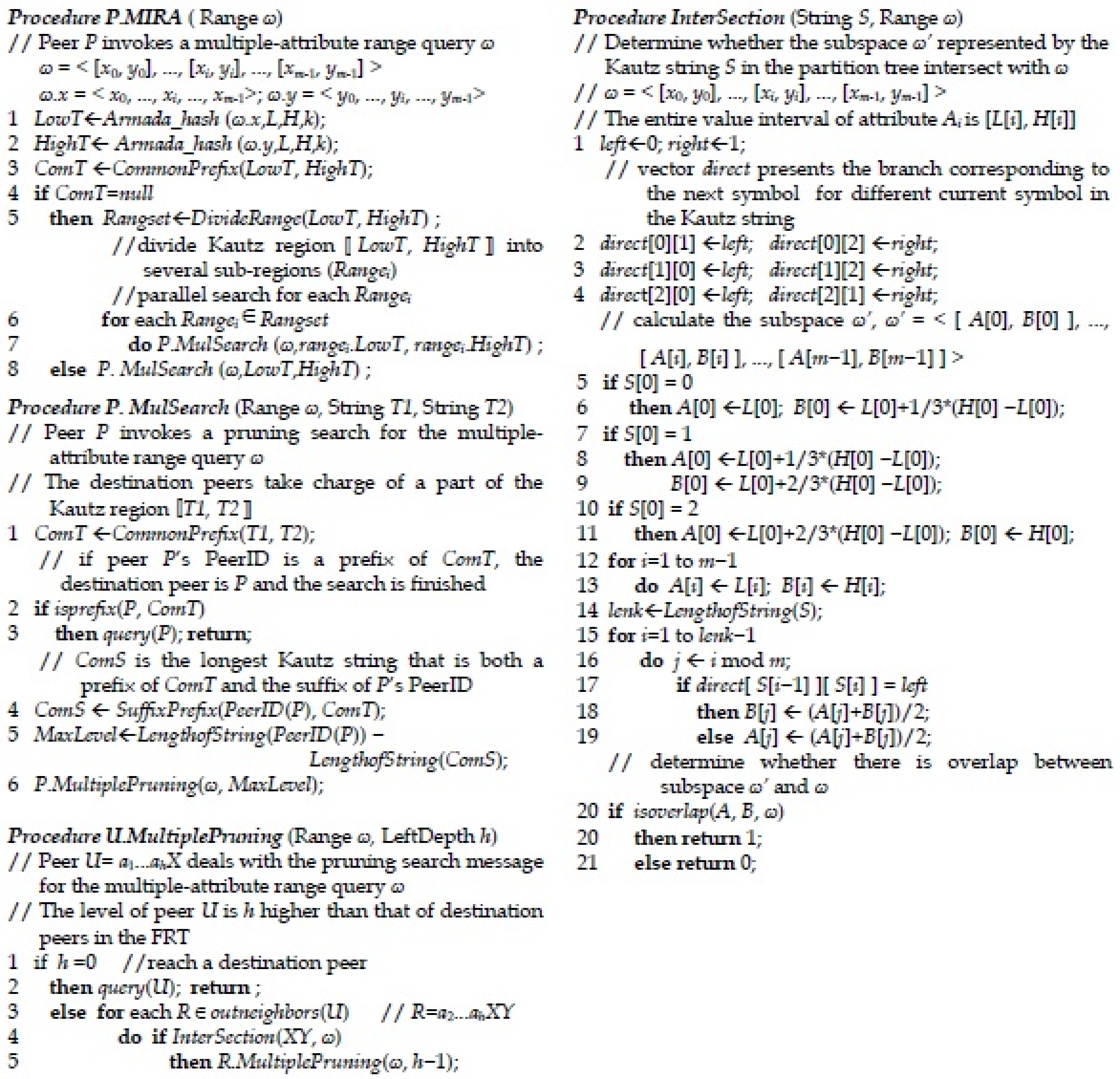
```
Procedure Multiple_hash (AttriValue V, MinAttri L,
    MaxAttri H, Length k)
// Generate an ObjectID (a Kautz string S of length k)
// The multiple-attribute value of the object is V
// The value of attribute $A_i$ of the object is V[i] and the
   entire value interval of attribute $A_i$ is [L[i], H[i]]
left←0; right←1; m←NumofAttri(V) ;
  //initialize the vector nextID
nextID[0][left] ←1; nextID[0][right] ←2;
nextID[1][left] ←0; nextID[1][right] ←2;
nextID[2][left] ←0; nextID[2][right] ←1;
for i=0 to m−1
  do A[i] ← L[i]; B[i] ← H[i];
  // value V lies in the subspace represented by the first
     child of the root node
if V[0] ∈ [ L[0], L[0]+1/3*(H[0] −L[0]) ]
  then S[0] ←0;
       A[0] ←L[0]; B[0] ← L[0]+1/3*(H[0] −L[0]);
  // value V lies in the second child of the root node
if V[0] ∈ ( L[0]+1/3*(H[0] −L[0]),
                      L[0]+2/3*(H[0] −L[0]) ]
   then S[0] ←1;
        A[0] ←L[0]+1/3*(H[0] − L[0]);
        B[0] ← L[0]+2/3*(H[0] − L[0]);
  // value V lies in the third child of the root node
if V[0] ∈ ( L[0]+2/3*(H[0] − L[0]), H[0] ]
   then S[0] ←2;
        A[0] ←L[0]+2/3*(H[0] − L[0]);
        B[0] ← H[0];
  // determine whether value V is in the subspace
     represented by the left or right child
for i=1 to k−1
   do j ← i mod m;
     if V[j] > (A[j]+B[j])/2
       then direction←1;
            A[j] ← (A[j]+B[j])/2;
       else direction←0;
            B[j] ← (A[j]+B[j])/2;
     S[j] ←nextID[S[j − 1]][direction];
return S;
```

Figure 4.4: The pseudocode of the *Multiple_hash* algorithm (from [17]).

4.2.1.2 MIRA Algorithm

It is straightforward to prove that the *Multiple_hash* algorithm is a multiple attribute partial order preserving function from multiple attribute space to the $KautzSpace(2, k)$. When a peer publishes a range query ω = <$[x_0, y_0], ..., [x_i, y_i], ..., [x_{m-1}, y_{m-1}]$>, Armada uses a multiple attribute range query processing algorithm called MIRA to forward queries to appropriate peers. Suppose that δ_1 = <$x_0, x_1, ..., x_{m-1}$> and δ_2 = <$y_0, y_1, ..., y_{m-1}$>. We use the *Multiple_hash* algorithm to find the Kautz strings corresponding to objects δ_1 and δ_2 and call them LowT and HighT, respectively. It is clear that for each object O in range

ω, $\delta_1 \lhd O$ and $O \lhd \delta_2$. The *Multiple_hash* algorithm is a multiple attribute partial preserving function. So the range ω is a subset of the Kautz region $[\![LowT, HighT]\!]$. For example, if $\omega = <[1.2, 1.8], [1, 5]>$ is the range query, then $\delta_1 = <1.2, 1>$ and $\delta_2 = <1.8, 5>$. We use the partition tree shown in Figure 4.3 to obtain Kautz strings corresponding to δ_1 and δ_2. Thus, $[\![LowT, HighT]\!] = [\![0120, 0210]\!]$. This Kautz region contains five leaf nodes P, R, W, S and M. Based on the partial order preserving property of the *Multiple_hash* algorithm, all the nodes that intersect a query are *among* five Kautz region nodes, but they may not be adjoining leaf nodes. Figure 4.3 shows that W and S leaf nodes don't intersect the query. So to process a range query, we need an algorithm to forward the query to *only* those peers which intersect the query. MIRA is the algorithm proposed in [18] to forward a range query to the appropriate peers. Figure 4.5 shows the pseudocode of MIRA.



```
Procedure P.MIRA ( Range ω)
// Peer P invokes a multiple-attribute range query ω
   ω = < [x_0, y_0], ..., [x_i, y_i], ..., [x_{m-1}, y_{m-1}] >
   ω.x = < x_0, ..., x_i, ..., x_{m-1}>; ω.y = < y_0, ..., y_i, ..., y_{m-1}>
1  LowT←Armada_hash (ω.x,L,H,k);
2  HighT← Armada_hash (ω.y,L,H,k);
3  ComT ←CommonPrefix(LowT, HighT);
4  if ComT=null
5    then Rangeset←DivideRange(LowT, HighT) ;
             //divide Kautz region [ LowT, HighT ] into
               several sub-regions (Range_i)
             //parallel search for each Range_i
6          for each Range_i ∈ Rangeset
7            do P.MulSearch (ω,range_i.LowT, range_i.HighT) ;
8    else P. MulSearch (ω,LowT,HighT) ;

Procedure P. MulSearch (Range ω, String T1, String T2)
// Peer P invokes a pruning search for the multiple-
    attribute range query ω
// The destination peers take charge of a part of the
    Kautz region [T1, T2 ]
1  ComT ←CommonPrefix(T1, T2);
   // if peer P's PeerID is a prefix of ComT, the
      destination peer is P and the search is finished
2  if isprefix(P, ComT)
3    then query(P); return;
   // ComS is the longest Kautz string that is both a
       prefix of ComT and the suffix of P's PeerID
4  ComS ← SuffixPrefix(PeerID(P), ComT);
5  MaxLevel←LengthofString(PeerID(P)) −
                                  LengthofString(ComS);
6  P.MultiplePruning(ω, MaxLevel);

Procedure U.MultiplePruning (Range ω, LeftDepth h)
// Peer U= a_1...a_hX deals with the pruning search message
   for the multiple-attribute range query ω
// The level of peer U is h higher than that of destination
   peers in the FRT
1  if h =0    //reach a destination peer
2    then query(U); return ;
3    else for each R ∈ outneighbors(U)     // R=a_2...a_hXY
4           do if InterSection(XY, ω)
5                then R.MultiplePruning(ω, h−1);

Procedure InterSection (String S, Range ω)
// Determine whether the subspace ω' represented by the
    Kautz string S in the partition tree intersect with ω
// ω = < [x_0, y_0], ..., [x_i, y_i], ..., [x_{m-1}, y_{m-1}] >
// The entire value interval of attribute A_i is [L[i], H[i]]
1  left←0; right←1;
   // vector direct presents the branch corresponding to
        the next symbol  for different current symbol in
        the Kautz string
2  direct[0][1] ←left;  direct[0][2] ←right;
3  direct[1][0] ←left;  direct[1][2] ←right;
4  direct[2][0] ←left;  direct[2][1] ←right;
   // calculate the subspace ω', ω' = < [ A[0], B[0] ], ...,
        [ A[i], B[i] ], ..., [ A[m−1], B[m−1] ] >
5  if S[0] = 0
6     then A[0] ←L[0];  B[0] ← L[0]+1/3*(H[0] −L[0]);
7  if S[0] = 1
8     then A[0] ←L[0]+1/3*(H[0] −L[0]);
9          B[0] ← L[0]+2/3*(H[0] −L[0]);
10 if S[0] = 2
11    then A[0] ←L[0]+2/3*(H[0] −L[0]);  B[0] ← H[0];
12 for i=1 to m−1
13    do  A[i] ← L[i];  B[i] ← H[i];
14 lenk←LengthofString(S);
15 for i=1 to lenk−1
16    do  j ← i mod m;
17        if direct[ S[i−1] ][ S[i] ] = left
18          then B[j] ← (A[j]+B[j])/2;
19          else  A[j] ← (A[j]+B[j])/2;
   //  determine whether there is overlap between
        subspace ω' and ω
20 if  isoverlap(A, B, ω)
20    then return 1;
21    else return 0;
```

Figure 4.5: The pseudocode of the MIRA algorithm (from [17]).

In FissionE, the out-neighbours of peer $P = u_1 u_2 ... u_b$ are in the form of $P = u_2 u_3 ... u_b v_1 ... v_q$ with $0 \leq q \leq 2$. For demonstrating the routing path from a query issuer to a destinations peer, the *forward routing tree* (FRT) is defined in [18]. Each node of the FRT is a peer of FissionE. The root node is a query issuer. Children of each node are its out-neighbours which are sorted in increasing order from left to right. The number of levels in a FRT is $b + 1$. Figure 4.6 shows an example of the FRT.

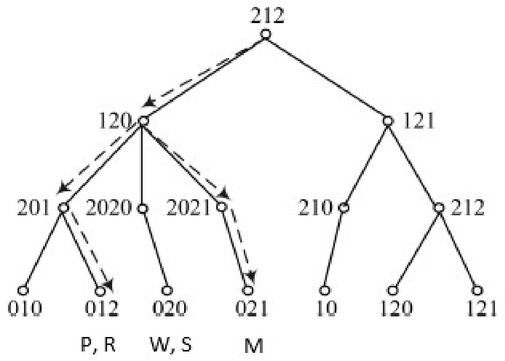


Figure 4.6: An example of the FRT and MIRA search path (from [18]).

To process a range query, we first obtain the Kautz region $[\![LowT, HighT]\!]$. If the Kautz string $LowT$ and $HighT$ don't have a common prefix, we need to divide the Kautz region into several subregions. In the case that $LowT$ and $HighT$ have a common prefix, we consider $ComT$ as the longest common prefix of $LowT$ and $HighT$. Then we get $ComS$ with length f as longest Kautz string which is both the prefix of $ComT$ and the root peerID (see line 4 of P.MulSearch in Figure 4.5). All the peers that intersect query ω are at the level of $b - f$ of the FRT. At level i of the FRT, the peer $B = u_{i+1}u_{i+2}...u_{b-f}X$ have some children in the form of $C = u_{i+2}u_{i+3}...u_{b-f}XY$. When B receives the query, it forwards the query to the children with their corresponding XY having an intersection with range query ω. Line 4 of algorithm U.MultiplePruining (which calls the InterSection algorithm) handle the sending of query messages only to peers intersecting the query range.

In Figure 4.6, an example of processing query by MIRA is shown. The dashed arrows shows the search path. The root peer 212 issues the range query $\omega = <[1.2, 1.8], [1, 5]>$. As explained before, for this query $LowT = 0120$ and $HighT = 0210$. So $ComT = 0$, $ComS = "null"$ and $f = 0$. In the result, all the destination peers are in level 3. In the node 120 of FRT, we calculate the subspace stored on peer 2020 and figure out that the subspace don't intersect with ω. Therefore the query is not forwarded to 2020.

4.2.1.3 Performance

The performance of Armada is evaluated by analysis and simulation in [18]. Armada is built on top of the FissionE which is a constant degree DHT. That means we need to store a constant number of pointers per peer. It is proven that the average number of messages to process one-dimensional queries in Armada is $\log N + 2n - 2$ where N is number of peers and n is the number of peers that intersect with the query. In addition, the maximum query delay for single and multiple attribute queries is less than $2 \log N$ hops in Armada. Query delay is the number of hops we need to traverse from the query issuer to detect the peer containing query results.

In [27], it has been shown the lower bound on the diameter of a constant-degree graph is $O(\log n)$. Based on this, it has been proven in [18] that the lower bound on the message cost of general range query schemes on constant-degree distributed hash tables (DHTs) is $O(\log n) + m - 1$. The complexity of Armada in 2-dimensional space is $O(\log n + m)$ in average where n is the number of nodes and m is the number of nodes that intersect with the query. For simplicity, Li et al [18] assume that the number of required messages for reporting points in their intersection to the query issuer is constant for each node. The worst case complexity in 2 or more dimensional space range queries on Armada is still an open problem.

4.2.2 Bisadi and Nickerson Distributed Spatial Data Structure

Bisadi and Nickerson in [5] use the non-redundant rainbow skip graph explained in the previous chapter to implement a distributed orthogonal range search in 2-dimensional space. The presented scheme is based on the non-redundant rainbow skip graph, so the topology of the proposed structure and message routing algorithm is based on the non-redundant rainbow skip graph.

To distribute the data on-to nodes of a non-redundant rainbow skip graph, the entire space is split into regions based on one of the coordinates (x or y). Then, each region is assigned to one node and the lower x coordinate bound of the region is considered to be the node key. Distribution based on x coordinates provides a total order relation for the key set. Figure 4.7 shows a distribution of 2-dimensional points among non-redundant rainbow skip graph nodes. In this figure, L_2 and U_2 are the lower and upper bound of the second node region and L_X, L_Y, U_X, U_Y are the range queries bounds.

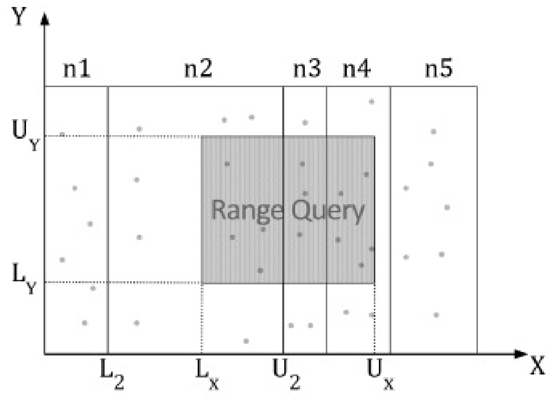


Figure 4.7: A distribution of a 2D space among 5 nodes (from [5]).

After distributing the data among nodes, the corresponding non-redundant rainbow skip graph is created. To search the non-redundant rainbow skip graph for range $\omega([L_x, U_x], [L_y, U_y])$, the non-redundant rainbow skip graph search algorithm is used to find the node whose region contains L_x. This node reports the results to the query issuer and forwards the query to the successor node if U_x is greater than the upper bound of its region. This step is continued until all the nodes which intersect with the query ω report the appropriate points.

Chapter 5

The SD-RTree

The *Scalable Distributed Rtree* (SD-Rtree) is a distributed spatial data structure supporting point, window and kNN queries proposed in [7]. The structure of an SD-Rtree is a distributed balanced binary tree that is constructed by splitting overloaded servers and insertion of new storage servers. The SD-Rtree aims at minimizing the number of servers. In other words, the structure adds or removes a server based on the size of the dataset. It assumes that queries are issued from a client node. There is an image (possibly incomplete) of tree structure on the client node which is used to address the tree for a specific query. In the case when the image is outdated, routing is done by forwarding the query among the servers. In the following sections we explain routing using an image in detail and discuss building the SD-Rtree and query algorithms.

5.1 Overall Structure of SD-RTree

The purpose of the SD-Rtree is constructing a cluster of nodes that provide fast access to a dataset of spatial objects. In addition, the data storage and query processing are evenly distributed on the servers. In an SD-Rtree each 2-dimensional object of a data set has an id (oid). The SD-Rtree is a binary tree which is mapped to a set of servers. There are two kinds of nodes in the tree: Routing node and data node. The properties of an SD-Rtree are as follow:
- each internal node is a routing node and has exactly two children.
- each routing node maintains left and right *directory rectangles* (*dr*) which are the minimal bounding boxes (mbb) of the left and right subtrees.
- each leaf node is a data node that stores a subset of objects.
- the height of the subtrees of each node differs by at most one.

The SD-Rtree has N leaves and $N-1$ internal nodes distributed over N servers. Each server S_i $(0 \leq i \leq N-1)$ contains one data node d_i and one routing node r_i except S_0 that contains only the data node d_0. Figure 5.1 shows an example of an SD-Rtree. In 5.1(a), we have one server containing all the data. After splitting, in 5.1(b) the whole data is distributed among S_0 and S_1, with S_1 storing r_1 and d_1. It is shown in Figure 5.1(b) that the directory rectangle of r_1 is $a = mbb(b \cup c)$. As mentioned in the properties of the SD-Rtree, a, b and c are kept in r_1 to guide insertion and search operations. Figure 5.1(c) shows a further split on server S_1. Figure 5.2 shows updating of nodes in the servers during split operations of Figure 5.1.

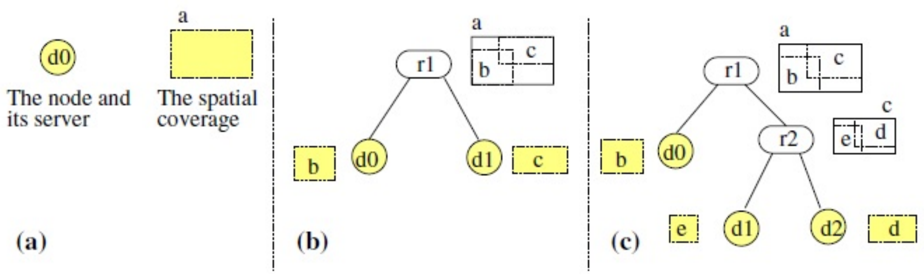


Figure 5.1: An example of SD-Rtree (from [7]).

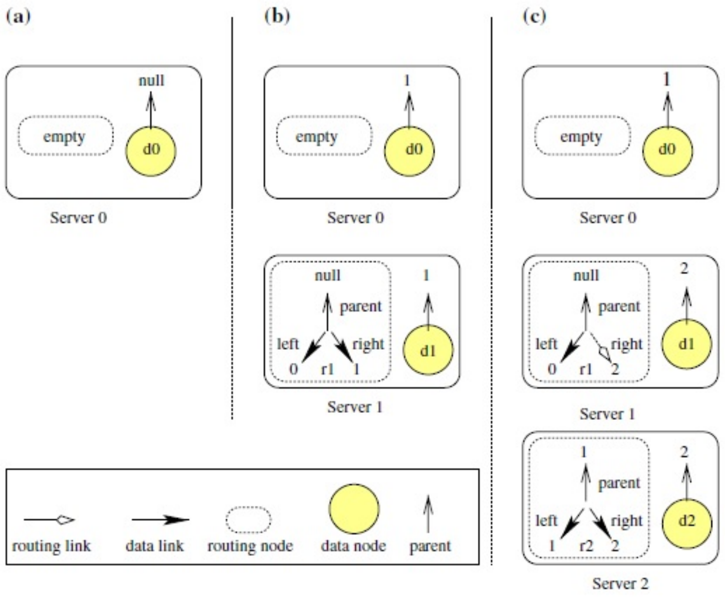


Figure 5.2: Updating of nodes in servers (from [7]).

5.1.1 Structure of an Internal Node

Each internal node in the SD-Rtree is a routing node. The routing node contains a local description of the tree with the following information:
- *height*: the height of the node that is $\max(left.heigth, right.heigth) + 1$.
- *dr*: the directory rectangle of the node.
- *left*, *right*: the links to the left and right children of a node. A link includes the id of the server that stores the reference node, the directory rectangle of the referenced node, the height and type of the referenced node.
- $Parent - id$: id of parent routing node.
- *OC*: the overlapping coverage which is an array that contains a shared part of the directory rectangle with other servers (see next paragraph).

Having overlap between directory rectangles of different nodes means that each node doesn't contain all the objects covered by its *dr*. To define a formal definition of overlapping coverage, first we explain some concepts. Let $N_1, N_2, ..., N_n$ be the ancestors of node N. Each ancestor N_i has two children, one of them is an ancestor of N or N itself and the other one is called the *outer* node $outer_N(N_i)$. For instance the outer node $outer_{d_2}(r_2)$ and $outer_{d_2}(r_1)$ in Figure 5.1 are d_1 and d_0, respectively.

The *overlapping coverage* or *OC* is defined as an array $OC_N = [1 : oc_1, 2 : oc_2, ..., n : oc_n]$ that oc_i is $N.dr \cap outer_N(N_i)$. In other words, *OC* stored on node N contains the intersections of N with outer nodes of the ancestors of N.

5.1.2 Structure of a Leaf Node

Each leaf node of an SD-Rtree is a data node containing the following information:
- *data*: the local data set stored on the server where the node resides.
- *dr*: the directory rectangle.
- *parent_id*: id of parent routing node.
- *OC*: the overlapping coverage.

It is worth noting that each node is identifiable by its type (data or routing) and the id (number) of its server.

5.1.3 The Image

As mentioned in previous sections, one of the principal properties of an SD-Rtree is maintaining an image on the client node. The image is a collection of links stored on the client node to provide a partial or outdated view of the tree structures. To process a query, the estimation about the address of target server (that contains data we are looking for)is done by the image. When the image is outdated, an incorrect server is chosen for routing the query. In this case, the structure forwards the query among servers and finally delivers the query to the correct server. Then, the correct server sends an *Image Adjustment Message* (IAM) to the client node to update the image.

Each time a server is visited, the following four links are collected:
- the data link describing the data node of the server,
- the routing link describing the routing node of the server,
- the left and right links describing the left and right subtrees of the routing node.

These four links are added to any message forwarded by the server, and finally all connected links are sent to the client as an IAM.

The rationale behind use of an image in routing is balancing the processing load. Usually, the servers that store the routing nodes located at or near the root receives many more messages. To distribute query processing among all servers, the image is used to estimate the location of the target server.

5.2 Tree Search and Updates

5.2.1 Insertion

To insert an object *o* with rectangle *mbb* to the SD-Rtree, the image on the client node has to be searched. Firstly, we find all the image data links whose directory rectangles contain *mbb*. Among such data links, the data link with smallest directory rectangle is chosen. If no such data link is found, we consider all routing links and choose the one whose height is minimum. In the case when we have more than one candidate, the link with the smaller directory rectangle is chosen. If we cannot find a link with above conditions, the data link whose directory rectangle is the closet to *o.mbb* is chosen.

After choosing the link, if the type of link is data, an INSERT-IN-LEAF message is sent to the server S containing the chosen link. If the type of link is routing, an INSERT-IN-SUBTREE message is sent to the server S.

When a server S receives the INSERT-IN-LEAF message, it compares the directory rectangle of its data node d_S with *o.mbb*. If d_S contains *o.mbb*, the server S inserts o into its repository; else the server sends an INSERT-IN-SUBTREE message to the parent of d_S.

When a server S' receives an INSERT-IN-SUBTREE message, it first compares the directory rectangle of its routing node $r_{S'}$ with *o.mbb*. IF d_S contains *o.mbb*, the top-down R-tree insertion algorithm [10] is carried out from $r_{S'}$; else the server forwards the INSERT-IN-SUBTREE message to the parent of $d_{S'}$.

As explained before, when the insertion procedure is done in more than one hop, the server inserting the new object sends back an IAM containing all collected links to the client node. Then the client node updates its image. Figure 5.3 shows the HANDLEMESSAGE insertion procedure.

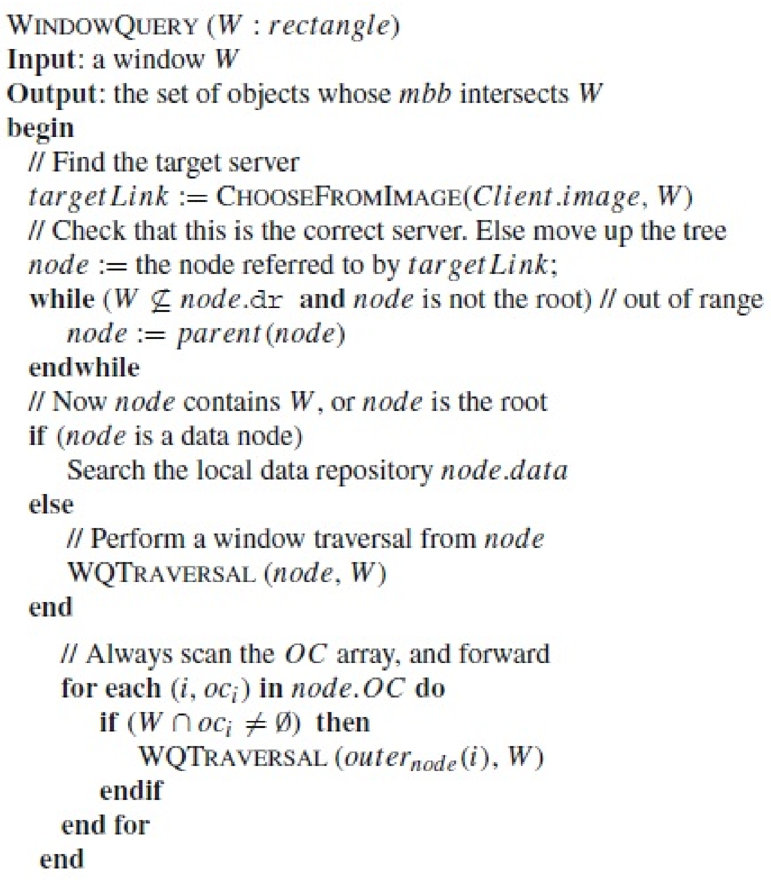
```
HANDLEMESSAGE ($C$ : $client$, $m$ : $Message$, $o$ : $object$)
Input: an object that must be inserted
Output: the appropriate action, depending on the message's type
begin
  // Add the links stored on the current server to the message
  Add {$data\_link$, $left\_link$, $right\_link$, $local\_link$} to $m.orp$
  switch ($m.type$)
  begin
    case INSERT- IN- LEAF:
      if ($o.mbb \subseteq current$.dr)
        // $o$ can be inserted in the current server
        Insert $o$ in the current server repository
        Update the local OC
        // If the server capacity is exceeded: split
        if ($current.nb\_objects > current.capacity$) then
          SPLITANDADJUST($current$)
        endif


        Send the IAM message to $C$
      else
        // Out-of-range: forward to the parent
        $m.type$ := INSERT- IN- SUBTREE
        Send $m$ to the server parent
      endif
      break
    case INSERT- IN- SUBTREE:
      if ($o.mbb \subseteq current$.dr or $current$ is the root)
        // Insert in the subtree rooted at $current$
        Go to the choice TOP- DOWN- TRAVERSAL
      else
        // Continue to forward the message bottom-up
        Send $m$ to the parent server
      endif
      break
    case TOP- DOWN- TRAVERSAL:
      // Let $l$ and $r$ be the left and right links of the current
      // server's routing node. Apply the CHOOSESUBTREE
      // algorithm which returns the inner and outer nodes
      ($I$, $O$) := CHOOSESUBTREE ($l.dr$, $r.dr$, $o.mbb$)
      // Enlarge the directory rectangle of $I$
      $I.mbb = I.mbb \cup o.mbb$
      if ($I$ is a data link) then
        // The chosen child is the leaf where $o$ must be inserted
        $m.type$ = INSERT- IN- LEAF;
        Send $m$ to the server $I.id$
      else
        // Compute and maintain the overlapping coverage
        if ($I.mbb \cap O.mbb$ has changed) then
          // Update the OC in the outer node
          UPDATEOC ($O$, $I$, $I.mbb \cap O.mbb$)
          // Update the OC in message
          $m.Rc$ := [$O.id$, $I.mbb \cap O.mbb$]
        endif
        // Recursive insertion message
        Send $m$ to the server $I.id$
      break
  end
end
```

Figure 5.3: The pseudo code description of the HandleMessage insertion algorithm (from [7]).

The cost of inserting one object into an existing SD-Rtree is one message on average. If the client sends the message to a node whose directory rectangle doesn't contain *o.mbb*, then in the worst case it takes $\log N$ messages to forward an insertion message to the root, where N is the number of servers (i.e. nodes). Another $\log N$ messages are necessary to find the appropriate data node for the Rtree insertion algorithm. Another $\log N$ messages are necessary to do splitting, if the server is overloaded by a new object. We need $3 \log N$ messages in the worst case. The worst case may occur when the image is outdated.

5.2.2 Searching

In order to process a window query W in a SD-Rtree, first the client node searches its image to find the link to a node that contains W. Then the client sends a message containing rectangle W to the server whose hosts the node. If the contacted node covers W, the classical Rtree window query algorithm WQTraversal is carried out from the node as well as overlapping coverage area. If the window W is out of contacted node range, the query message is forwarded to the contacted node parent. The window query algorithm which is called WindowQuery is shown in Figure 5.4 in detail.



```
WINDOWQUERY ($W$ : $rectangle$)
Input: a window $W$
Output: the set of objects whose $mbb$ intersects $W$
begin
  // Find the target server
  $targetLink$ := CHOOSEFROMIMAGE($Client.image$, $W$)
  // Check that this is the correct server. Else move up the tree
  $node$ := the node referred to by $targetLink$;
  while ($W \not\subseteq node$.dr and $node$ is not the root) // out of range
      $node$ := $parent(node)$
  endwhile
  // Now $node$ contains $W$, or $node$ is the root
  if ($node$ is a data node)
      Search the local data repository $node.data$
  else
      // Perform a window traversal from $node$
      WQTRAVERSAL ($node$, $W$)
  end
    // Always scan the $OC$ array, and forward
    for each ($i$, $oc_i$) in $node.OC$ do
        if ($W \cap oc_i \neq \emptyset$) then
            WQTRAVERSAL ($outer_{node}(i)$, $W$)
        endif
    end for
   end
```

Figure 5.4: The pseudo code description of WINDOWQUERY algorithm (from [7]).

5.2.3 Deletion

To delete an object in SD-Rtree, we need to define the constant τ which is the percentage of full capacity for a server. First, using the WINDOWQUERY procedure explained above, the object o is found and deleted. After deletion, based on the value of τ for the server S hosting the object o, three cases are possible as follows:

1. the space occupancy of sever S is not below τ. In this case, after deletion of object o, it is necessary to adjust the directory rectangles of the nodes by a bottom-up tree traversal. In the worst case, we need to adjust the *dr* of all the nodes in the path from server S to the root. In addition, the OC of the nodes whose directory rectangle is adjusted, must be updated.

2. the space occupancy of sever S is below τ but the space occupancy of its sibling S' is strictly above τ. After deletion, S' sends some of its objects to S that reduces

the spatial overlapping between two siblings. S and S' both update their directory rectangle and OC, and send update message to their parent like previous case.

3. the space occupancy of sever S is below τ and the space occupancy of its sibling S' is τ.

In this case both servers must be merged. Objects of S are inserted to S' and server S will be free. Data node stored on S' becomes the child of its grandparent. In addition, we need appropriate adjustment of the height of nodes in the tree.

Chapter 6

Apache Hadoop

Hadoop is an open source software framework that provides a reliable shared storage and analysis system. The hadoop framework contains different modules and capabilities. The two most fundamental parts of hadoop are MapReduce and HDFS. Hadoop Distributed File System or HDFS is distributed file system that provides shared storage of data on a cluster of machines. MapReduce is a programming model that performs processing analysis of data.

6.1 MapReduce Model

MapReduce is a programming model for processing of large datasets. In this programming model, the user defines two functions, *map* and *reduce* to provide a specific computation in a parallel manner. In the following subsection, the programming model is explained in detail.

6.1.1 Programming Model

In the MapReduce model, to process and analyse the data, the user needs to specify the query as two distinct phases, the map phase and the reduce phase. The map function and reduce function have to be defined by programmer. The map function takes a key-value pair as input and produces a set of intermediate key-value pairs. In this step all intermediate pairs containing the same key are grouped and passed to the reduce function. The reduce function each time accepts an intermediate key and all corresponding values for that key, and combines these values to produce usually one key-value pair as output. The following example from [26] shows a sample of map and reduce functions.

Consider the problem of finding the maximum global temperature for each year in a large dataset. The input of map phase in this example is raw data. Each record of raw data contains different information fields collected by weather stations. In this example, the map function works like a preprocessing phase. It extracts the year and the air temperature of each record and drops the records with missing values. The output of the map function is as follows:

$(1950, 0)$
$(1950, 22)$
$(1950, -11)$
$(1949, 111)$
$(1949, 78)$
The MapReduce framework processes the map phase output, and then passes it as input to the reduce phase. The processing past preceding reduce consists of grouping and sorting the key-value pairs by the key. The input of the reduce phase is as follows:
$(1949, [111, 78])$
$(1950, [0, 22, -11])$
The reduce function goes through the list of values for each key, and takes the maximum temperature for each year. The final output of the reduce function is as follows:
$(1949, 111)$
$(1950, 22)$
The logical data flow of the above example is shown in Figure 6.1.

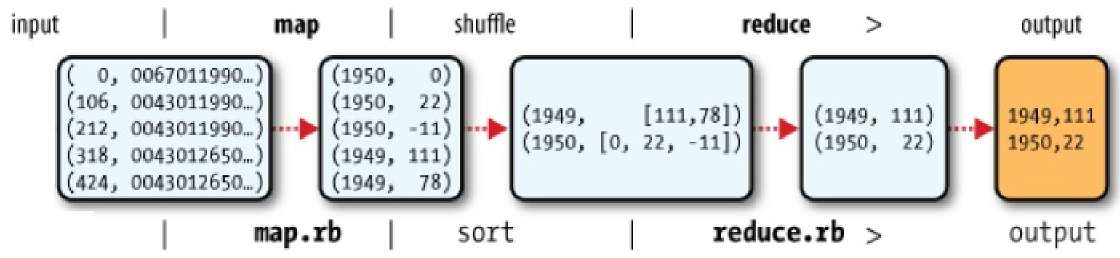


Figure 6.1: An example of MapReduce logical data flow (from [26]).

6.1.2 Execution Overview

In MapReduce execution, the input data is automatically partitioned into M splits. These splits of data can be processed by different machines in parallel. The intermediate keys produced by the map phase are partitioned into R splits. The partitioning can be performed by a user defined partitioning function. The default partitioning function is $hash(key)$ mod R which places keys in buckets using a hash function.

In Figure 6.2 the overall data flow of MapReduce is illustrated.

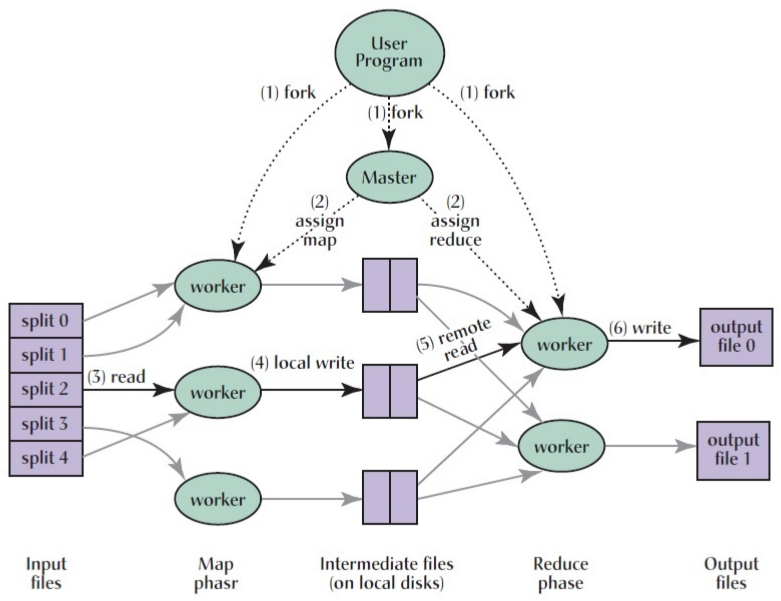


Figure 6.2: MapReduce execution overview (from [6]).

Firstly the user program calls the MapReduce function, and then the following actions occur:

1. The input file is partitioned into M pieces, each one of size $16 - 64MB$ (controllable by the user). Then many copies of the map program start working on a cluster of machines in parallel.
2. One of the MapReduce program copies is called the master program and the other ones are workers. To execute a program, M map tasks and R reduce tasks have to be divided by the master among workers. The master program assigns a map task or a reduce task to idle worker programs.
3. The map workers read their corresponding input data split. It parses the key-value pair, passes each key-value pair to the map function and produces the intermediate key-value pairs. The intermediate key-value pairs are buffered in memory.
4. The buffered pairs are written to the local disk periodically and then partitioned into R splits. The locations of these R splits are sent back to the master program. The master program forwards these location to the reduce workers.
5. The reduce workers use the received location to read the buffered intermediate key-value corresponding to its partition. Then the reduce workers sort data by their intermediate keys and groups together the data with the same key. The reduce workers use an external sort, if necessary.

6. The reduce workers pass each intermediate key and its corresponding list of values to the reduce function, and appends the result to a final output file that belongs to this reduce function.
7. After completing all map and reduce functions, the master worker reports back to the user program.

After the above execution steps, there are R output files available. Users typically don't need to combine the files. They are usually passed to the another MapReduce program or another distributed application.

6.1.3 Refinements and Performance

A few extensions are added to the MapReduce programming model,as follows: [6]

- As explained in the previous section, a user can define a new partitioning function to map the intermediate keys to the R reduce workers.
- Within each partition, the intermediate key-value pairs are sorted by key in increasing order.
- To decrease the data transfer between map and reduce tasks and save available bandwidth, the user can specify a combiner function. The combiner function combines the output of map tasks with the same key within the same map task. This reduces the amount of work done in reduce tasks.
- The MapReduce library supports different types of input and output data.
- The MapReduce library provides a single machine mode for debugging and testing of the program.

The performance of MapReduce in [6] is measured by two computations. Each one of these computations is representative of a large subset of programs written by MapReduce users. The first computation scans approximately one terabyte of data to look for a specific rare pattern. This problem is the representative of a class of program that extracts a small amount of interesting data from a large dataset. In this example, $M = 15000$ and each split is about $64MB$ and $R = 1$. In other words, the entire output is in one file.

Figure 6.3 shows the rate of input data scanning over time for the above example. As the number of machines assigned to the MapReduce tasks increases, the rate of scanning increases and the peak occurs when we have 1764 workers. When the map tasks finish, the rate starts dropping. The computation including a minute of startup overhead takes 150 seconds totally. The startup overhead is due to distribution of programs to all worker machines, and opening input files in the GFS (Google File System).

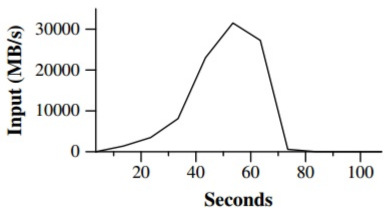


Figure 6.3: Data transfer rate over time (from [6]).

6.1.4 Large-Scale Indexing

MapReduce in [6] is used to implement an indexing system as a data structure for the Google Web search service. The input of the system is a large set of documents stored as a set of GFS files. The MapReduce implementation of an indexing system is composed of a sequence of 8 MapReduce operations. The advantages of using MapReduce are as follows [6]:

1. The MapReduce code is simpler and smaller. In MapReduce indexing code, the code for distribution of data, fault tolerance and parallelization is hidden in the MapReduce library.

2. In MapReduce indexing, all unrelated tasks are kept separate. So it is easy to modify a specific part of code.

3. In the MapReduce indexing process, performance can be improved easily by adding more machines. In addition, it doesn't need an operator to respond to machine failures or network problems. The MapReduce library automatically deals with such problems.

6.2 The Hadoop Distributed Filesystem

The Hadoop Distributed File System (HDFS) is a distributed file system for data storage used by hadoop applications. Although HDFS has some similarities with other distributed file systems, many specific characteristics make it distinct from others. HDFS highly tolerates failure of machines and can be deployed on low cost hadware. In addition, HDFS provides high throughput access to data. A number of goals of HDFS are as follows [1]:

- Due to the huge number of machines in the network, there is a non-trivial probability of machine failure. Fault tolerance is an important goal of HDFS performed by detecting faults followed by quick and automatic recovery.

- HDFS needs to store large data sets. Scalability is another important property of HDFS to reliably store and process massive amounts of data.

- It is more efficient to execute a requested computation near the data it operates on. When the size of the data is huge, it is better to move the computation closer to where the data is located rather than moving the data to where application is running. HDFS does this to decrease network congestion.
- HDFS is easily portable from across heterogeneous hardware and different operating systems.
- HDFS provides reliability by automatically maintaining multiple copies of data and automatically redeploying processing logic in the event of failures. The number of copies is 3 by default and can be modified by user.

6.3 Hadoop Cluster

A cluster of machines in hadoop supports three different modes:

- Local or Standalone mode: This non-distributed mode is he tdefault configuration of hadoop in which hadoop runs in a single Java process. This mode is used for debugging.
- Pseudo-distributed mode: Although pseudo-distributed mode runs on a single node, it mimics the operation of a real cluster. In this mode, each hadoop daemon runs in a separate Java process.
- Fully-distributed mode: this mode is the real production cluster.

As the map and the reduce code in hadoop will be running on a large volume of data distributed among hundreds of nodes, any bugs may be amplified and make system-wide problems. Using the three modes and iterative development in hadoop, a user is able to remove bugs on a small data set before deploying on a distributed cluster.

The cluster machines in hadoop are classified into three groups: Master nodes, Slave nodes and Client machines. Master nodes supervise two essential functions in hadoop, storage of data and analysis; Master nodes have two types: Name Node and Job Tracker. The name node and job tracker are responsible for coordinating and controlling of data storage in HDFS and parallel processing of data, respectively. The majority of nodes in hadoop clusters are slave nodes. The salve nodes run a data node and a task tracker daemon. The daemons on slave nodes communicate with their master node to receive instructions. The task tracker is a slave to the job tracker, and the data node daemon is a slave to the name node.

The client machines load data into the cluster and submit MapReduce jobs describing how that data should be processed, and then view the results of the job when it is finished. Figure 6.4 describes briefly different types of nodes in a cluster.

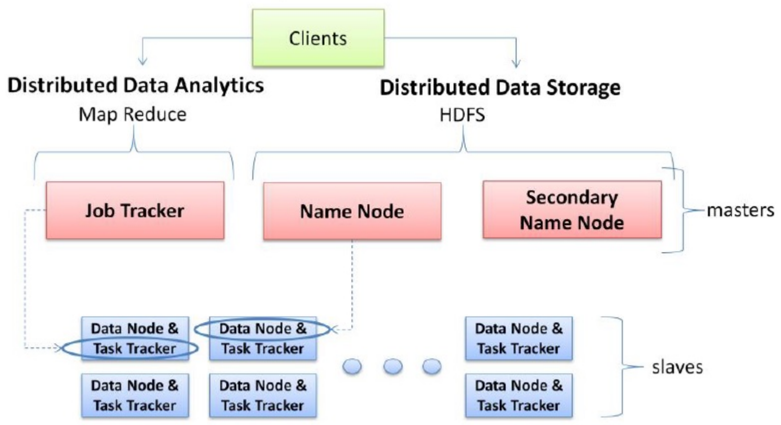


Figure 6.4: Different roles in hadoop (from [11]).

Some recent research [22, 23] used MapReduce as a framework for implementing and analysis of indexing strategies. None of this research, however, has implemented distributed spatial indexing to support range queries on MapReduce. The possibility of using MapReduce to process multidimensional range queries while minimising data transfer between nodes is still an open problem.

Chapter 7

Comparison

In chapters 4 and 5, three distributed spatial data structures are discussed. To our knowledge these are the only distributed data structures proposed for spatial data indexing. In this chapter we aim to compare these three data structures, Armada [18], Bisadi and Nickerson data structure [5] and SD-Rtee [7].

Before summarizing the complexity of these three data structures, we introduce a number of variables and symbols, as follows:

- n: The number of nodes or peers in the network.
- K: The number of points reported in range.
- B: The number of reported points that fit in one message.
- α: The fraction of points reported in range; $\alpha \in [0, 1]$, with $K = \alpha N$ where N is the number of points.
- $|S_i|$: The number of points stored in node i.
- m: The number of nodes intersecting the query Q.
- $S(N)$: The memory size of the data structure including number of data items and pointers.
- $Q(N, j)$: Worst case j-dimensional range search cost in number of messages passed.
- $I(N, d)$: The cost of inserting one new point in an existing data structure storing N points in number of messages.
- $D(N, d)$: The cost of deleting one new point in an existing data structure storing N points in number of messages.

Table 7.1: Comparison of Armada, SD-Rtree and Bisadi and Nickerson data structures.

Parameters	Armada	Bisadi and Nickerson	SD-Rtree
$S(N)$	$O(n+N)$	$O(n+N)$	$O(n+N)$
$Q(N,1)$ [a]	- [b]	$N.A.$	-
$Q(N,2)$	-	$O(n+\frac{K}{B})$	$O(\log n)$
$Q(N,d)$	-	$N.A.$	$N.A.$
Expected Cost	$O(\log n + 2m - 2)$ [c]	$O(\log n + n\sqrt{\alpha} + \frac{K}{B})$ [d]	$O(\log n)$ [e]
$I(N,d)$	$O(\log n)$	$O(\log n)$	$3\log n$
$D(N,d)$	$O(\log n)$	$O(\log n + \frac{\lvert S_i \rvert}{B})$	-
Supports Redundancy	*no*	*yes*	*no*
Peer-to-peer	*yes*	*yes*	*no*
Can index rectangle	*no*	*no*	*yes*

[a] The cost of reporting points in the query is counted in the search cost of Bisadi and Nickerson but this is not counted in Armada and SD-Rtree.

[b] Dash shows an open question for corresponding data structure.

[c] This is the average message cost for 1-dimensional range search by Armada. No theoretical proof for d-dimensional range search is presented in [18].

[d] Expected cost is computed for 2-d range queries.

[e] This is for 2-dimensional space.

Chapter 8

Open Problems

This report present a survey of recent developments on spatial data structures supporting orthogonal range search on a distributed computing model. We explored the advantages and disadvantages of these data structures. We identified several open problems, as follows:

- Is there a linear space distributed spatial data structure that supports optimal worst case $O(\log n + m + \frac{K}{B})$ messages for $Q(N, 2)$ range search in peer-to-peer network?
- Is there a data structure with redundancy permitting any one node to be off-line and still answer any 2-d range query? If so, what is the worst case space and messaging cost of this data structure? What about 3-d data?
- Is there an adaptive grid structure that can support efficient dynamic distributed spatial data structures in a peer-to-peer networks?
- What is the $Q(N, d)$ cost of MIRA and PIRA in Armada in the worst case?
- Is there any way to implement range query processing in 2 or more dimensional space on a MapReduce framework? If so, how efficient is MapReduce indexing running on hadoop clusters in comparison with other distributed spatial data range query processing schemes?

Bibliography

[1] Apache hadoop. `http://hadoop.apache.org/`, 2013.

[2] Peyman Afshani, Lars Arge, and Kasper Dalgaard Larsen. Orthogonal range reporting in three and higher dimensions. In *Foundations of Computer Science, 2009. FOCS'09. 50th Annual IEEE Symposium on*, pages 149–158. IEEE, 2009.

[3] Pankaj K. Agarwal. Range searching. *Handbook of Discrete Computational Geometry*, 1997.

[4] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms (TALG)*, 3(4):37, 2007.

[5] Pouya Bisadi and Bradford G Nickerson. Orthogonal range search using a distributed computing model. In *CCCG*, 2011.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Cédric Du Mouza, Witold Litwin, and Philippe Rigaux. Large-scale indexing of spatial data in distributed repositories: the sd-rtree. *The VLDB JournalThe International Journal on Very Large Data Bases*, 18(4):933–958, 2009.

[8] Michael T Goodrich, Michael J Nelson, and Jonathan Z Sun. The rainbow skip graph: a fault-tolerant constant-degree distributed data structure. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 384–393. ACM, 2006.

[9] Michael T Goodrich, Michael J Nelson, and Jonathan Z Sun. The rainbow skip graph: A fault-tolerant constant-degree p2p relay structure. *arXiv preprint arXiv:0905.2214*, 2009.

[10] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching*, volume 14. ACM, 1984.

[11] Brad Hedlund. understanding hadoop clusters and the network. *Studies in Data Center Networking, Virtualization, Computing*, 2010.

[12] Mostafa Abd-El-Barr Hesham El-Rewini, editor. *Advanced Computer Architecture and Parallel Processing*. Wiley-Interscience, 1st edition, 2005.

[13] Joseph O'Rourke Jacob E. Goodman, editor. *Handbook of Discrete Computational Geometry*. CRC Press, 2nd edition, 2004.

[14] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2008.

[15] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to parallel computing*, volume 110. Benjamin/Cummings Redwood City, 1994.

[16] Ching Law and K-Y Siu. Distributed construction of random expander networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 2133–2143. IEEE, 2003.

[17] Dongsheng Li, Jiannong Cao, Xicheng Lu, and Kaixian Chan. Appendix of TKDE paper "efficient range query processing in peer-to-peer systems". `http://doi.ieeecomputersociety.org/10.1109/TKDE.2008.99/`. [Online; Jan-2009].

[18] Dongsheng Li, Jiannong Cao, Xicheng Lu, and Kaixian Chan. Efficient range query processing in peer-to-peer systems. *Knowledge and Data Engineering, IEEE Transactions on*, 21(1):78–91, 2009.

[19] Dongsheng Li, Xicheng Lu, and Jie Wu. Fissione: A scalable constant degree and low congestion dht scheme based on kautz graphs. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, volume 3, pages 1677–1688. IEEE, 2005.

[20] Marc van Kreveld Mark Overmars Mark de Berg, Otfried Cheong. *Computational Geometry*. Springer, 3rd edition, 2008.

[21] Jiří Matoušek. Geometric range searching. *ACM Computing Surveys (CSUR)*, 26(4):422–461, 1994.

[22] Richard McCreadie, Craig Macdonald, and Iadh Ounis. Mapreduce indexing strategies: Studying scalability and efficiency. *Information Processing & Management*, 48(5):873–888, 2012.

[23] Richard MC McCreadie, Craig Macdonald, and Iadh Ounis. Comparing distributed indexing: To mapreduce or not? *Proc. LSDS-IR*, pages 41–48, 2009.

[24] J Erickson Pankaj K. Agarwal. Geometric range searching and its relatives. In *Advances in Discrete and Computational Geometry*. Citeseer, 1999.

[25] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[26] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[27] Jun Xu, Abhishek Kumar, and Xingxing Yu. On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks. *Selected Areas in Communications, IEEE Journal on*, 22(1):151–163, 2004.