2013

Partial Replica Location And Selection For Spatial Datasets

Yun Tian
University of Mississippi

PARTIAL REPLICA LOCATION AND SELECTION FOR SPATIAL DATASETS

A Dissertation
presented in partial fulfillment of requirements
for the degree of Doctor of Philosophy
in the Department of Computer and Information Science
The University of Mississippi

by

YUN TIAN

July 2013

ABSTRACT

As the size of scientific datasets continues to grow, we will not be able to store enormous datasets on a single grid node, but must distribute them across many grid nodes. The implementation of *partial* or *incomplete* replicas, which represent only a subset of a larger dataset, has been an active topic of research. *Partial Spatial Replicas* extend this functionality to spatial data, allowing us to distribute a spatial dataset in pieces over several locations.

We investigate solutions to the partial spatial replica selection problems. First, we describe and develop two designs for an *Spatial Replica Location Service (SRLS)*, which must return the set of replicas that intersect with a query region. Integrating a relational database, a spatial data structure and grid computing software, we build a scalable solution that works well even for several million replicas.

In our SRLS, we have improved performance by designing a R-tree structure in the back-end database, and by aggregating several queries into one larger query, which reduces overhead. We also use the *Morton Space-filling Curve* during R-tree construction, which improves spatial locality. In addition, we describe *R-tree Prefetching(RTP)*, which effectively utilizes the modern multi-processor architecture.

Second, we present and implement a fast replica *selection* algorithm in which a set of partial replicas is chosen from a set of candidates so that retrieval performance is maximized. Using an R-tree based heuristic algorithm, we achieve *O(n log n)* complexity for this NP-complete problem.

We describe a model for disk access performance that takes filesystem prefetching into account and is sufficiently accurate for spatial replica selection. Making a few simplifying assumptions, we present a fast replica selection algorithm for partial spatial replicas. The algorithm uses a greedy approach that attempts to maximize performance by choosing a collection of replica subsets that allow fast data retrieval by a client machine. Experiments show that the performance

of the solution found by our algorithm is on average always at least 91% and 93.4% of the performance of the optimal solution in 4-node and 8-node tests respectively.

ACKNOWLEDGEMENTS

I express my deepest gratefulness to my advisor, Dr. Philip J. Rhodes, who has given me the opportunity to be his research assistant, my first crucial step towards my career goal. I am very grateful for his earnest and generous support and help during my graduate study. I feel I am quite fortunate to have this considerate and helpful advisor, who constantly helps me to grow professionally and academically. I thank my committee members, Dr. Gregory L. Easson, Dr. Byunghyun Jang and Dr. Feng Wang for their time and valuable comments, making my dissertation more solid and more complete.

I appreciate the support and the assistantship provided by the Department of Computer and Information Science at the University of Mississippi. Particularly, I thank Dr. H. Conrad Cunningham for many suggestions and help he has given to me. I am very thankful to other colleagues and professors at Olemiss, who have helped me in my teaching and my research. Lastly, this thesis is dedicated to my wife and my two sweet children.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

MOTIVATION

As the size of datasets continues to grow, it is becoming more expensive and challenging for researchers to generate, process and store entire datasets locally. Grid and Cloud computing attempt to address this problem by coordinating geographically distributed computational resources, and by replicating complete datasets to make them available at multiple sites [1, 2, 3, 4, 5].

There are several limitations to the traditional approach. If a dataset is sufficiently large, copying a replica over a network can take several days. Such a delay means replication cannot be conducted in a spontaneous or ad hoc manner, but must instead be performed well in advance of the scheduling of tasks. The scheduling process itself suffers as well. Computation is typically scheduled on computational resources next to the data, even when such resources are scarce. If an insufficient number of replicas exist, tasks may be delayed until new replicas are constructed.

The most troubling limitation is imposed by the size of local resources. If we require an entire dataset to be stored at a single site, we are limited by the size of local storage. Even if capacity is plentiful at other sites, we cannot use it unless we are allowed to break the dataset up into smaller pieces. Therefore, the maximum feasible problem size is constrained by the capacity of the site with the greatest local storage. Furthermore, if computation is only performed on machines local to the data, then even if local storage is sufficient, local computational resources could still place a ceiling on maximum problem size.

The obvious solution to the storage problem is to allow the creation of partial replicas, in which only a portion of the original dataset is copied to a new location [6, 7, 8]. To address the computation problem, we might allow computation to be scheduled on machines that are not local to the data required by the computation. This option is particularly attractive when a computation requires only a small portion of a larger dataset, keeping transfer costs low.

Such a scheme requires efficient access to *subsets* of a larger dataset or replica. It also requires a replica selection method that can help to satisfy a data query from several different partial replicas. Replica selection and scheduling decisions could then be made in advance of the data access itself, boosting performance.

Quite often, replicas management and selection are left to the user's discretion. Many distributed Object-based storage systems are developed, such as Parallel Virtual File System (PVFS) [9], Panasas [10] and Lustre [11]. These systems store data as a collection of objects of variable size. Other systems such as Google File System [12], Hadoop Distributed File System [13] and OpenStack [14] provide infrastructure for cloud computing and storage. It is the user's responsibility to manage complete files and choose among them for the entire dataset, though the dataset may be divided into chunks and the system may manage replicated chunks transparently.

For a motivating application scenario, assume we have a gigantic piece of 3D spatial dataset, covering the entire North American continent. Each point in the dataset has a longitude value, latitude value, an altitude value and other attributes. How should we store and share this dataset effectively and in a timely manner? How should we develop an effective approach to interactively visualize a region of interest?

We divide the whole data into smaller pieces according to their geographical region. Dividing data into smaller pieces suggests two advantages. First, distributed storage system becomes scalable for the larger dataset. Second, data processing becomes more parallel. For example, Memphis TN and Chicago IL each is described by a separate partial replica. When accessing data for Memphis, we only visit the nodes that host the partial replicas for Memphis. The problem is that the Memphis data might be copied and reorganized and stored on other grid or cloud nodes. Then before retrieving data, we have to choose which replica we should access in order to minimize the distributed transfer time. Therefore, effective replica selection algorithm could improve performance for out-of-core interactive visualization in a distributed system [15][16].

Under some circumstances, scientists commonly share and publish their array-based data by leaving them in a curating data center. Curating center immensely improves data security,

availability and consistency. Besides, it reduces the cost associated with data management for science and engineering. The curating center manages replication, backup and recovery etc. The curating data center also provides services for scientists to identify and to retrieve a subset of interested datasets. The proposed Spatial Replica Location Service and Partial Replica Selection approach are very useful in the curating data center, such as the Open Data project at NASA [17].

However, as datasets continue to grow at an unprecedented rate, we envision that the size of such datasets eventually surpasses the storage capacity at one storage site. For example, NASA currently provides billions of gigabytes (an Exobytes) of data from its rich history of planetary, lunar, terrestrial, and Earth-orbiting missions [18]. In this case, our proposed system and techniques are more suitable for a geographically distributed system. Our Partial Replica Location Service and Data Server are scalable in its design, which could alleviate the storage limits at many levels, including memory level, disk level and storage at one site. The feature of subset accessing enables to process large datasets in a piece-wise fashion.

In the future, we will extend our system to incorporate both the unstructured GIS datasets and the array-based datasets on one platform. For instance, during climate simulations, a climatologist attempts to find an optimal and safe route for valley residents to evacuate. The simulation involves array-based climate data and unstructured or structured GIS data. The simulation should examine many paths on the map, and decide whether the route is safe or flooded during the evacuation. We expect our system to provide very useful supports for this type of computation.

CHAPTER 2

INTRODUCTION

Replication and *Replica Selection* are widely used in distributed systems to help distribute both data and the computation performed upon it. The *Globus Toolkit* includes a *Replica Location Service(RLS)* that provides users with a flexible mechanism to discover replicas that are convenient for a particular computation [1][19][20][21][22]. By providing multiple copies of the same dataset, replicas can increase both I/O bandwidth and options for scheduling computation.

Recently, we have been working on the problem of partial replica selection for *spatial* data. A spatial dataset associates data values with locations in an n-dimensional domain and is commonly used in various fields of engineering and the sciences, often as a product of simulation. For example, a climatologist might use spatial data produced by simulation to predict temperature changes for the Gulf of Mexico over a long time period. An agronomist might use spatial data to represent soil humidity for a large tract of land. For another example, a climatologist might use a regular grid to simulate the behavior of the atmosphere over the Gulf Coast for hurricane prediction, or an electrical engineer might calculate the magnetic field surrounding an antenna. The size of spatial datasets is growing rapidly due to advances in measuring instrument technology and more accessible but less expensive computational power.

The *Geographical Information Systems (GIS)* community has been actively investigating distributed data access for some time [23][24][25]. GIS data generally refers to the surface of the earth, and is usually two dimensional. Information such as road locations, municipal boundaries, bodies of water, etc., can be represented using a sequence of points denoting the feature. Although this type of data is certainly spatial, our own research is focused more on *volumetric* data. For example, *Computational Fluid Dynamics (CFD)* simulations commonly represent three or four

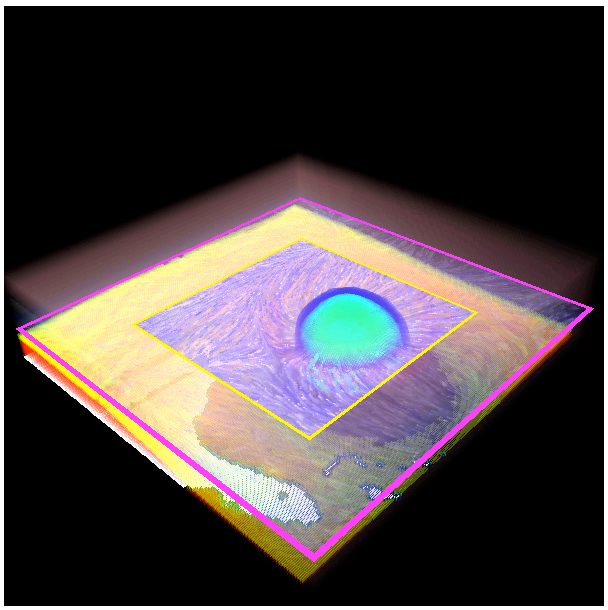


Figure 2.1: Hurricane Isabel image from the Visualization 2004 contest entry of Schafhitzel, Weiskopf and Ertl.

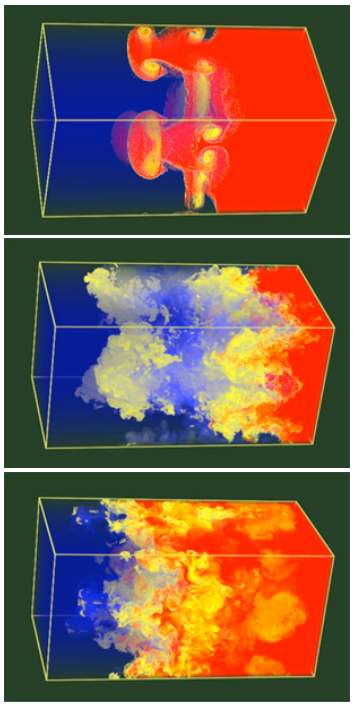


Figure 2.2: Physics simulation of *Richtmyer-Meshkov* instability, image produced by the ASCI team at LLNL courtesy of https://computation.llnl.gov/casc/asciturb/simulations.shtml.

dimensional volumes using rectilinear grids of data points that span the volume. Conceptually, such datasets (and subsets extracted from them) often have the shape of a rectangular prism, while GIS data may have an entirely irregular shape.

In figure 2.1 and 2.2, we show two spatial datasets produced by climate simulation and physics simulation respectively. Figure 2.1 presents the hurricane Isabel data with a total size of around 60 Gigabytes, which consists of many $500 \times 500 \times 100$ dense arrays. Each point in the space domain has thirteen variables, such as pressure, moisture and temperature etc.. The dataset consists of 48 time steps. Figure 2.2 shows three of time steps in Richtmyer-Meshkov instability simulation, with a space domain of $192 \times 192 \times 448$ and a total size of 2 Terabytes.

Replication has long been used to increase data availability and performance in grid computing environments, and the problem of *replica selection* has been addressed by many researchers [1, 2, 26, 27, 28]. However, replica selection for spatial data is less well understood, mainly because of the special difficulties presented by spatial datasets.

If a dataset is stored on disk in *linear* order, a spatial subset query corresponds to a large number of small accesses in the one dimensional file space. For example, a request for a 100^3 subset of a larger volume would correspond to 10,000 separate accesses to the file. Replica selection for spatial datasets requires a model of access time that incorporates both disk and network costs, allowing the replica with suitable performance to be selected.

Partial replicas are incomplete copies of the original dataset. They may represent only a few of the original attributes, or could be a spatial subset of the original volume. They could be produced in advance, or as the by-product of user access to other copies of the data. If some areas of a dataset are more useful and interesting than others (i.e. they are *hot spots*), replication of these areas can increase availability and performance. We show three partial spatial replicas in figure 2.3, with each located at a different location within the entire space domain. In our system, each replica may have one or more copies that might be stored on different storage nodes. For example, the red replica in figure 2.3 could be copied on node one and node three, but with different ways of storing them on disk, called storage organization. Therefore, we must find a way to identify the

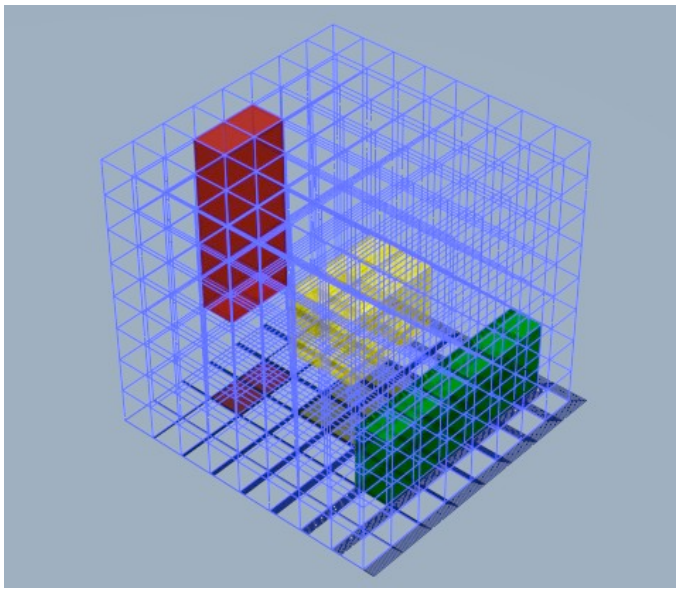


Figure 2.3: Examples of 3D partial spatial replicas in a space domain.

location and the spatial extent for each replica. We will not address these questions until chapter 3.

The work described here was performed within the context of the *Distributed Spatial Computation and Visualization Environment (DISCoVer)*. DISCoVer is a package of software that facilitates the visualization of large spatial datasets distributed in a grid or cloud. DISCoVer has three components: Granite, Magnolia and IDEA. IDEA is a Map-reduce like environment for spatial data [29]. The other two components are described in the following.

The *Granite* component of DISCoVer provides efficient access to spatial datasets stored on local or remote disks [30, 31]. Granite allows its users to specify spatial subsets of larger n-dimensional volumes for retrieval. It takes advantage of *UDT* [32, 33, 34, 35], a UDP based reliable data transfer protocol that is well suited to the transfer of large data volumes over high bandwidth-delay product connections. The combination of UDT and Granite provides fast access to *subsets* of datasets stored remotely. If a computation needs only a comparatively small subset of a larger volume, accessing that subset remotely may be much faster than moving the entire volume. The flexibility provided by this additional option is especially welcome in heterogeneous environments, where the hardware that best suits a computation (e.g. a GPU) might be located far

from the dataset.

The *Magnolia* component of the DISCoVer system is intended to integrate Granite's unique spatial capabilities with existing Grid software. Replica selection for partial spatial replicas is an important part of Magnolia, and requires both a way to discover the partial replicas that intersect with a spatial query and a model of access time that allows Magnolia to choose between them. The Granite system already incorporates the concept of a *storage model*, which can be used to describe the access pattern of a given query, then infer disk access costs.

Before a computation begins in a grid, partial spatial replica selection must address two subproblems. The first subproblem is to efficiently identify the set of partial replicas that intersect the subvolume required by the computation in a distributed system. Second, from that set, we want to select the "best" replicas to minimize transfer time. The latter requires a metric based on factors such as disk and network bandwidth and latency, data storage organization, etc. In order to solve these problems, we design a Spatial Replica Location Service (SRLS) and provide a Partial Replica Selection Algorithm.

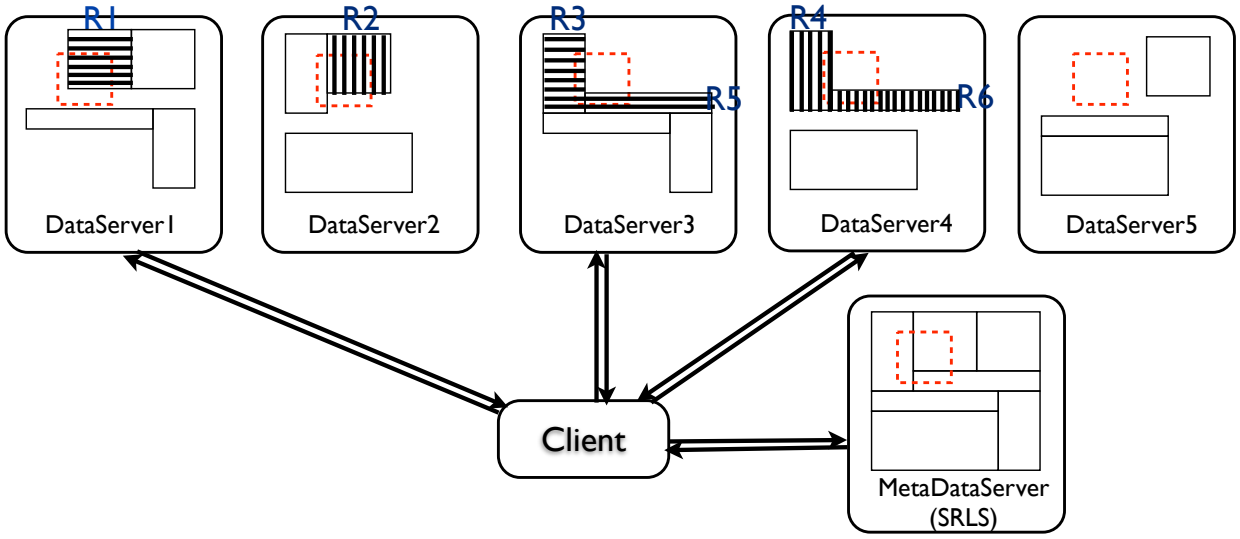


Figure 2.4: Design of our distributed storage system for partial spatial replicas.

Figure 2.4 shows the design of our distributed storage system for partial spatial replicas.

This structure resembles the design found in Gfarm file system [3], Lustre [11] and Globus Toolkit Data Management [36]. It takes two steps for a client program to access the distributed data. First, the client contacts a metadata server, in order to identify the replicas required by the computation. Second, after analyzing the metadata, the client then visits the relevant data servers where the requested replicas are stored. The metadata server maintains the descriptive data for each replica, including the spatial extent, physical address, size and modified time etc. In our system, we call the metadata server the *Spatial Replica Location Service (SRLS)*. In chapter 3 and chapter 4, we describe and verify two designs of the Spatial Replica Location Service.

Let us look at the details in figure 2.4. The subset query (shown as a red dotted square) intersects with replicas R1 through R6 on four data servers. We observe that R1 and R2, likewise, R3 and R4 have the same spatial extent, but with different storage organizations (row major or column major). Research shows that different storage organizations result in great difference in performance. Therefore, we should choose among them to maximize the performance. Our replica selection model and algorithm have to make a decision whether to retrieve data from R1 or R2 for the subset query. Data server two and data server five are not visited for this specific subset query. We present and evaluate our replica selection models and algorithm in chapter 5.

The proposed system and technology are intended to be quite useful in a system like Coastal Emergency Risks Assessment (CERA) [37]. Hurricane surge forecasting are models that forecast the overland surge during a hurricane and require the use of large spatial databases, for example, the Advanced Circulation and Storm Surge model (ADCIRC) and the SWAN Wave model. Data input to these models include topography and bathymetry of a large section of a coast, ADCIRC meshes and wind fields etc.

In these systems, scientists tend to run simulation models many times, but with different parameters or algorithms for each run. These models perform the same simulation task, but might be developed with different simulation approaches. In order to achieve high performance, simulations (computation) might be distributed and run simultaneously at different sites. But each simulation works with the entire data domain. In this case, the generated data is also distributed as

a result of distributed computation.

We observe two reasons for partitioning the entire dataset into incomplete replicas, the *Partial Spatial Replicas*. If we consider the United States as the entire data domain, it is rather prohibitive to present entire US in a thirty-meter resolution. For administrative purposes, regional authority or laboratories show more interests in the features in a local region. For example, a modeler in Mississippi might use a part of the Gulf of Mexico to model storm surge in the Mississippi Gulf Coast. Secondly, scientists create partial spatial replicas as a by-product of remote subset query. Sometimes, scientists copy a subset of the dataset to a local machine and then process them.

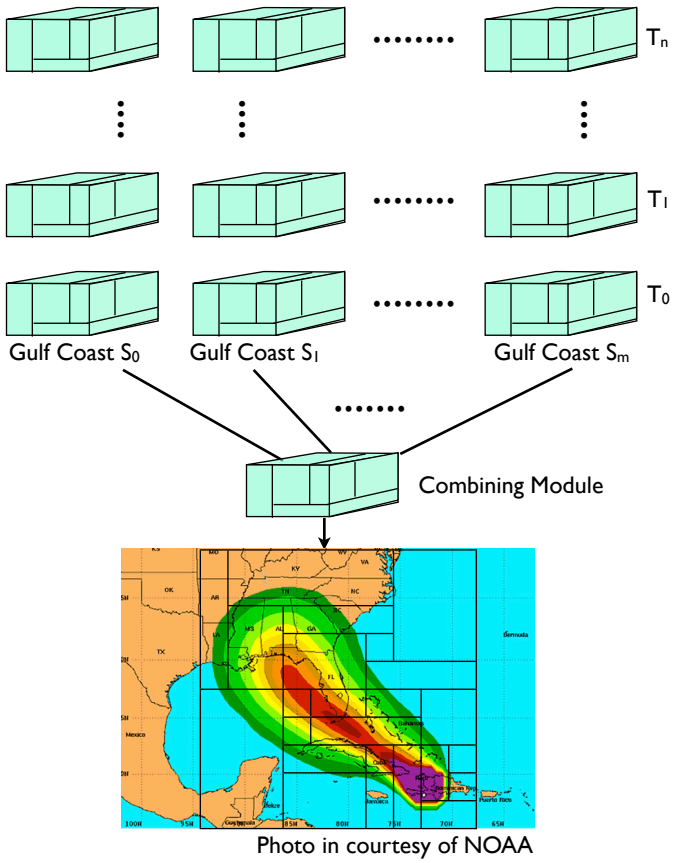


Figure 2.5: Application in hurricane tracking and overland surge.

Figure 2.5 illustrates an application where our system is very useful. To accurately predict the path of a hurricane, scientists execute m simulation models for Gulf Coast area, $S_0, S_1, ..., S_m$.

All models process the same space data domain, that is, the Guld Coast data. But they execute with different sets of parameters. To accelerate the simulation and prediction, they run the models concurrently in a distributed system. Each model has to simulate for n time steps, $T_0, T_1, ..., T_n$. After that, another module collects the simulation data from various distributed computers, and combine them to obtain the final prediction.

In short, our eventual goal is a system where both data and computation are truly distributed and the size of simulation is not constrained by the local capacity. Our system can work with both local and remote datasets, regardless of the placement of the data in the system. We also do not restrain the size and the shape of the partial replicas. Partial replicas could have arbitrary size and shape.

CHAPTER 3

PARTIAL REPLICA LOCATION SERVICE WITH THE GLOBUS TOOLKIT R-TREE

In this chapter, we first introduce the concept of *minimum bounding rectangle (MBR)* to represent spatial extent, then followed by how to quickly identify intersected replicas. We suggest that an R-tree is quite useful to accelerate replicas searching. The rest of this chapter is organized as follows. The Globus Toolkit RLS component is described and its limitations for spatial data management are analyzed in section 3.3. Then we present the GTR-tree and its implementation in section 3.4. Section 3.5 provides experimental results and analysis.

3.1 Representing Spatial Extent

In order to present spatial queries or denote the size of spatial replicas, we need a representation for volumes and sub-volumes in spatial datasets. This representation should be independent of the file format used by the underlying file. We chose an n-dimensional generalization of the *minimum bounding rectangle (MBR)* representation, also known as an *axis aligned bounding box (AABB)*. Because we assume that the sides of the box are aligned with the major axes, we only need to explicitly represent the two extreme corners of the box. Rotation is not allowed. The "low" corner consists of the smallest coordinate values for each axis, while the "high" corner consists of the largest coordinate values. For example, $\{(10,5,20),(15,10,25)\}$ is a three dimensional MBR denoting a $5 \times 5 \times 5$ cube. We can use this representation both to specify spatial queries and to denote the portion of the original dataset that each partial spatial replica represents.

3.2 Algorithm Overview

Before a computation begins in a grid, partial spatial replica selection must address two subproblems. The first subproblem is to efficiently identify the set of partial replicas that intersect

the subvolume required by the computation in a distributed system. Second, from that set, we want to select the "best" replicas to minimize transfer time. This requires a metric based on factors such as disk and network bandwidth and latency, data storage organization, etc. Chapter 3 and chapter 4 address the first subproblem. In chapter 5, we present several metric models and a simple but effective replica selection algorithm.

Figure 3.1 illustrates how the intersecting replicas are identified in the system. Spatial replicas and queries are commonly represented as MBRs, which approximate the extent of the object in space using minimum and maximum values for each dimension. In the 2D case, a spatial replica or query can be represented as $\{x_{min}, y_{min}, x_{max}, y_{max}\}$. These MBRs identify the region of the larger data space that each replica represents. Intersection tests are performed to determine which replicas intersect with the spatial query. In the worst case, each spatial replica in the catalog will be examined against the spatial query, which is very computationally expensive.

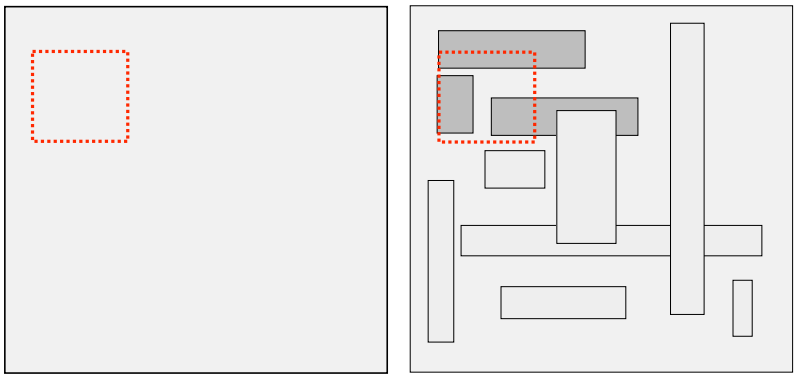
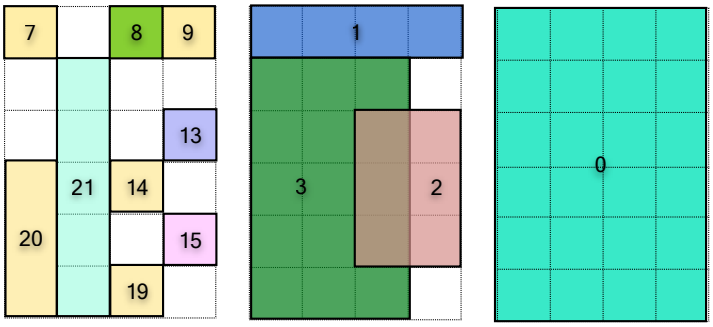


Figure 3.1: Intersected Replicas in 2D. The dotted square represents the spatial query, and intersected replicas are shown as shaded rectangles.

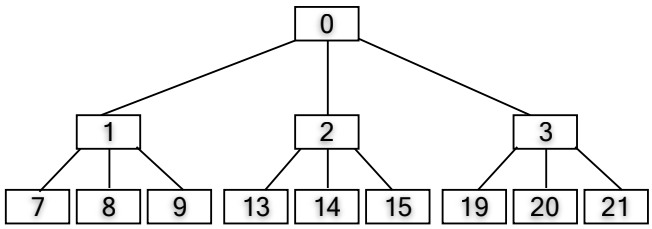
However, an *R-tree* can be used on each grid node to organize the metadata of all spatial replicas and to prune the search space. We define the *out degree* of a tree node as the actual number of child nodes. Each node of an R-tree can have a variable number of child nodes, up to a maximum value that we will refer to as the *fanout*. For our purposes, leaf nodes contain a reference to partial replicas, along with an MBR for each replica. Internal nodes contain references to child tree nodes, along with an MBR that encloses the MBRs of all the child nodes. During search, an

internal node's MBR can be tested for intersection with the query rectangle, allowing large sections of the tree to be pruned.

In figure 3.2, we show an example of R-tree with 2D spatial replicas. Here, R-tree groups nearby replicas *7, 8 and 9* and represents them with their MBR *(rectangle 1)* in the next higher level of the tree. Because all children of node 1 lie within the MBR of node 1, if a query does not intersect the MBR of node 1, it also cannot intersect any of the contained child MBRs. Then the searching algorithm just uses the MBR to decide whether or not to search inside a subtree. In this way, most of the nodes in the R-tree are never visited during a search.



(a) Layout of 2D *spatial* replicas packed into an R-tree of three levels. Root node is identified as tree node 0. The middle level of the R-tree consists of three nodes.



(b) R-tree with fanout = 6 and out degree = 3 for figure (a).

Figure 3.2: An example of R-tree with 2D spatial replicas.

We propose a technique, the *Globus Toolkit R-tree (GTR Tree)*, by combining the Globus Toolkit *Replica Location Service (RLS)* component with the R-tree, to speed up replica location service. The proposed GTR-tree is implemented on top of the Globus Toolkit RLS component, and

introduces the concept of "data management about data management". To facilitate the replicas selection, all the replicas are registered in the Local Replica Catalogue (LRC). The LRC employs a relational database to organize and manage the metadata of all replicas, with one database entry corresponding to one replica. Similar to traditional indexing techniques, we used the R-tree to reorganize these metadata entries in the relational database to make spatial queries more efficient. Due to the characteristics of the spatial data, i.e., multi-dimensional bounds, the traditional database indexing technique on separate axis is not helpful making the spatial query faster in a relational database.

3.3 Globus Toolkit RLS

The Globus Toolkit is an open source software toolkit used for building grids, a collection of computer resources from multiple locations to achieve a common goal. It is being developed by the Globus Alliance and many others all over the world [38]. Globus Toolkit consists of many components, including job management, grid security and data management etc.

The Globus Toolkit data management component consists of two major subcomponents: *GridFTP* and the *Replica Location Service (RLS)* [20, 39]. GridFTP is designed for fast data transfer in grid environments, while RLS provides a mechanism to register the existence of a replica and for clients to discover them. Specifically, clients specify a *logical file name (LFN)* to the RLS, which then returns a list of *physical file names (PFNs)*.

For example, the dataset with LFN "isabel-simulation" might have many full replicas in the grid, each identified using a PFN like "gsiftp://nodex.domain.org/tmp/data/isabel-1". In the more complicated case, the "isabel-simulation" dataset might have many partial replicas in the whole grid, each representing a subset of the original dataset. Here, we must take advantage of the Globus Toolkit's ability to represent user-defined attributes and associate them with LFNs or PFNs. The attribute type can be a *date*, *float*, *int*, or *string*, allowing considerable flexibility and expressiveness.

Figure 3.3 shows an example RLS configuration, in which RLS servers have been config-

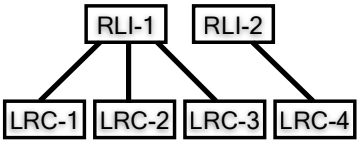


Figure 3.3: Example of RLS Configuration

ured as either an *RLI* or *LRC*. A *Local Replica Catalog (LRC)* keeps track of the files stored on a particular node. A *Replica Location Index (RLI)* coordinates the information from several LRCs, and provides a point of contact for client programs. The LRCs periodically send their own state to higher-level RLIs using *Soft State Update*, to inform RLIs of the list of the LFNs that the LRC manages. In particular, in figure 3.3, LRC-1, LRC-2, and LRC-3 send updates to RLI-1. When client programs query the RLI with a PFN, they will be referred to one or more applicable LRCs. The interested reader can find documents about RLS on the Globus website [22].

The back end of the RLS server is a relational database. An ODBC layer is used between the server and the database to provide support for databases from different vendors. The database contains a table for LFNs, a table for PFNs, mapping tables, a general attribute table, and four attribute tables for each attribute type, which are used to associate values with a PFN or LFN. The database design and the table structure is presented in [20].

3.3.1 Limitations of the RLS for Spatial Data

The Globus Toolkit provides the fundamental mechanism for replica discovery with great scalability and flexibility. However, it has limitations when it is used with spatial data. Spatial data objects are commonly represented by a *Minimal Bounding Rectangle(MBR)*, which approximates the extent of the object in space using minimum and maximum values for each dimension[40, 41]. For example, we denote the extent of a three dimensional object as $\{x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}\}$.

An *intersection query* identifies the spatial replicas which intersect with a given query rectangle. Intersection queries are not efficiently supported by the RLS, because such queries are not supported by the underlying relational database. We must therefore implement support for

this functionality outside of the database. In this work, we chose to implement this support outside of the RLS as well, using an unmodified Globus toolkit. An alternative approach is to implement spatial support within the RLS module, which is a topic for next chapter.

To determine if a particular replica intersects with the query rectangle, we must examine the values of the MBR in each dimension. Performing this intersection test can be expensive when dealing with large numbers of replicas, so we would like to minimize the number of times it is performed. Unfortunately, the relational database which supports the RLS has no facility for culling the set of replicas to be tested, so all replicas in the database must be tested with each new intersection query. An important contribution of this work is that we implement a data structure that vastly reduces the number of intersection tests required for each spatial query, producing dramatic gains in performance.

3.4 The Globus Toolkit R-tree

The R-tree was first invented by Guttman[41] for spatial data indexing, dynamic insertion and deletion; later many R-tree variants were proposed, attempting to improve R-tree performance using different tree construction strategies, such as the R*-tree[42], the Hilbert R-tree[43] and the Priority R-tree[44]. Meanwhile the R-tree was extended to the parallel R-tree, a structure allowing parallel spatial queries [45, 46, 40, 47]. In this work, the *packed R-tree* is described and used to demonstrate the proposed GTR-tree performance. The dynamic insertion and deletion from the GTR-tree will be investigated in future work.

The basic R-tree data structure uses the MBRs described in section 3.3.1. Each node can have a variable number of entries, up to a maximum value that we will refer to as *fanout*. For our purposes, leaf nodes contain a reference to partial replicas, along with an MBR for each replica. These MBRs identify the region of the larger data space that each replica represents.

Internal nodes contain references to child tree nodes, along with an MBR that encloses the MBRs of all the child nodes. During search, an internal node's MBR can be tested for intersection with the query rectangle, allowing large sections of the tree to be pruned.

3.4.1 GTR-tree Overview

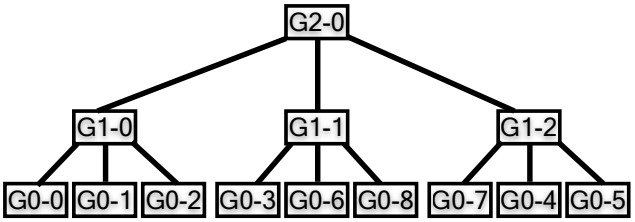
In this chapter, we combine the Globus RLS service described in section 3.3 with the R-tree data structure. In order to achieve this, four issues must be addressed. First, the spatial data objects are declustered onto the grid nodes, which maintain spatial metadata in their local RLS databases.

Second, the metadata entries in the database are packed using an R-tree packing algorithm, resulting in a single GTR tree root on each grid node.

Third, to minimize the network latency and inter-node communication, the spatial query is invoked on the remote grid node by using GRAM [48] to submit a spatial query job to many remote grid nodes. All of the remote grid nodes traverse an R-tree which is stored in the relational database, instead of in memory. The job submitter can schedule and dispatch independent parallel query jobs. We chose GRAM because the overhead of GRAM discussed in section 3.5 is tolerable for medium and large queries and paid very infrequently. More importantly, GRAM is built upon the *Grid Security Infrastructure(GSI)*, a secure, reliable and standard Grid tool, allowing users to deploy with ease.

Fourth, after the query completes on each grid node, three options are available for the query results. We can transfer results back to the client with GridFTP; We can do the same using a Java socket; We can leave results on the remote grid node for further processing. The performance of these options is presented in section 3.5. Similar arguments can be applied to the use of GridFTP as with GRAM.

In figure 3.4(a), an example R-tree is presented, where the notation in the rectangles represents the *id* of a tree node. To represent a GTR-tree in the Globus Toolkit RLS database, each node in the tree is described as a set $\{id, MBR, info\}$, where *id* is the reference to the node, *MBR* is a minimum bounding (hyper)rectangle that encloses all child *MBR*s, and $info$ contains references to the child nodes. The RLS back-end database, as mentioned in section 3.3, has a table named *t_str_attr* which contains three columns *obj_id, attr_id, value*. Column *obj_id* signifies the PFN id or LFN id in the database, *attr_id* is the attribute id defined in the RLS database, and column *value* in this table is used to store the string attribute value associated with the PFN or LFN. Each



(a) Example R-tree with fanout=3

Id	Attribute	Value
G0-0	MBR	“0,0,1,1”
G0-0	Info	“<gtrtree> <pfn>G0-0</pfn> <numChild>0</numChild> <Address>gsiftp://dragon.cs.olemiss.edu/data/d0.bin </Address> </gtrtree>”
……	……	……
G2-0	MBR	“0,0,4,6”
G2-0	Info	“<gtrtree> <pfn>G2-0</pfn> <numChild>3</numChild> <child0>G1-0</child0> <child1>G1-1</child1> <child2>G1-2</child2> </gtrtree>”

(b) Example GTR-Tree node representations for (a)

Figure 3.4: A GTR-tree Example on a single grid node. The first number in each tree node name indicates the level, while the second provides a unique identifier within that level. The Distributed GTR-tree structure is presented in figure 3.6.

GTR-tree node is represented as an entry in the RLS database, with most information stored in the *t_str_attr* table, and the PFNs used to represent the replica ids. By associating *MBR* and *info*, two user-defined string attributes, with the PFNs, a GTR-tree node can be described using two rows in the *t_str_attr* table. The *MBR* of each node uses the format mentioned in the previous section expressed as a string, while *info* uses an XML-based string to provide information concerning the number of children, physical address (if it is a leaf node), and child PFNs.

Figure 3.4(b) presents the *t_str_attr* table in the database, i.e., the corresponding GTR-tree for figure 3.4(a). The layout of the replicas for this GRT-tree is shown in figure 3.5. Note that all the partial spatial replica metadata is stored only in the leaf nodes. By taking advantage of the GTR-tree, it is not necessary to traverse all the entries in the *t_str_attr* table to answer the spatial query, so the search space is vastly reduced. The correspondingly dramatic effect on query performance is discussed in section 3.5.

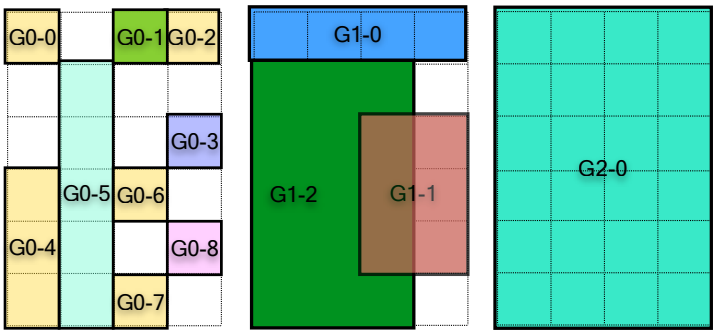


Figure 3.5: A 2-D packing example. As suggested by Beckmann et al., we try to minimize the overlap between nodes, as well as the empty space in each internal node. From right to left, subfigure corresponds to the first level (root level), second level and third level of the R-tree in figure 3.4(a).

There are two reasons to separate the *MBR* and *info* fields. First, when processing a spatial query, it is more efficient to directly retrieve the *MBR* from the database, rather than parsing this information out of the *info* XML string. Second, the *value* column in the *t_str_attr* table is limited to 250 characters by the Globus Toolkit, so moving *MBR* out provides more space in the *info* filed for child information, increasing tree fanout. Of course, for very high fanouts, more fields can be provided to represent child information.

3.4.2 Implementation

Our implementation requires declustering metadata among grid nodes, packing the data objects into tree nodes, and performing the spatial replica query itself. We discuss each of these below:

3.4.2.1 Spatial Replica Generation And Leaf Node Declustering

Different metadata declustering algorithms result in different parallel performance, although comparing many declustering algorithms is not the focus of this work. Among several approaches to declustering, such as *Random*, *Round Robin*, and *Proximity Index* [40], we chose an approach similar to Random and Round Robin, randomly assigning the synthesized spatial replicas to different nodes.

In order to evaluate our implementation, we needed to generate test cases consisting of large numbers of rectangles representing the spatial extent of partial replicas. Instead of randomly generating all spatial replicas at once, and then declustering and sorting, the spatial replicas are synthesized independently for each grid node in sorted order.

To accomplish this, we represent the rectangles in a $2n$ dimensional space defined by the coordinates of the low and high corners, similar to Arge, et al. [44]. For example, a single hyperpoint in a 6 dimensional conceptual space completely defines a rectangle's position and extents in 3D space. From an arbitrary starting point, we can weave a meandering path through the conceptual space by adding random vectors $\delta_0 \ldots \delta_{k-1}$ to each successive position, resulting in a sequence of k rectangles. By constraining the magnitude of each δ_i, we can ensure that the rectangle sequence demonstrates good locality in the data space, an important property for effective packing.

3.4.2.2 Packing Algorithm

Many packing approaches only differ in the first step, to sort the hyper rectangles in specific order, which determines how the replicas are grouped into the R-tree nodes [49]. It is considered an efficient packing if the resulting R-tree conforms to the four optimization criteria proposed by Beckmann et. al. [42].

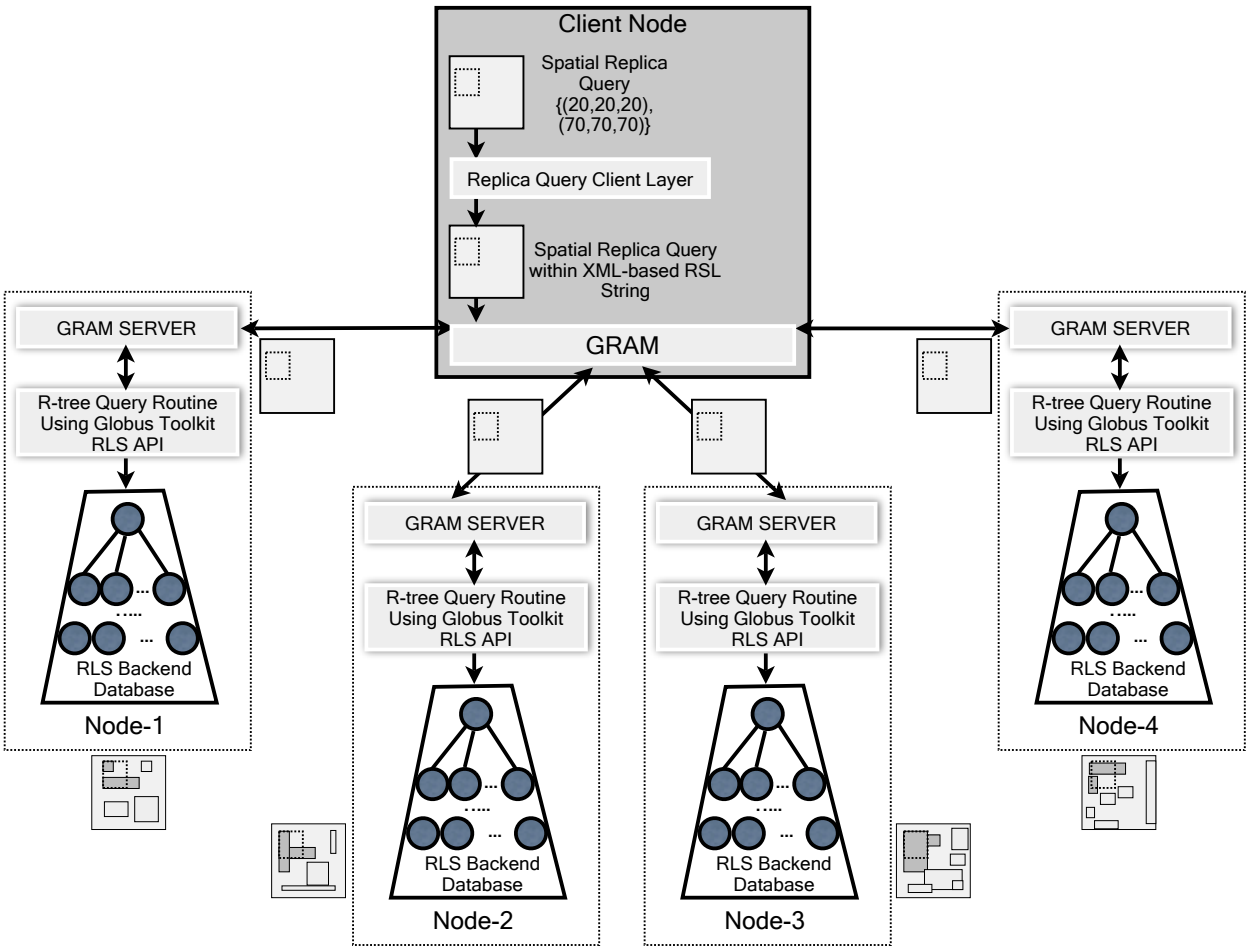


Figure 3.6: The Distributed GTR-tree for our experimental grid is distributed across an arbitrary number of nodes. In this diagram, the Local Replica Catalog (LRC) on each node represents a distinct subset of the available partial replicas. Any grid node can serve as the Client Node to accept the user's query. Once the query is submitted, the Client Node performs the query on its own LRC, as well as invoking the same query on the other nodes using GRAM.

In figure 3.5, a 2-D packing example using the greedy search for nearest hyper-point is provided and demonstrates how this method endeavors to minimize the overlap between R-tree nodes and the dead space of each node, two of the optimization criteria proposed for R-tree construction in [42]. In figure 3.5, the sorted list is {G0-0, G0-1, G0-2, G0-3, G0-6,G0-8,G0-7,G0-4,G0-5}, which results in a packed R-tree in figure 3.4(a) with fanout 3.

We use the general bottom-up packing algorithm [49] after the 3D replica metadata is synthesized in a sorted way. The replicas are initially written to a text file and the leaf nodes are grouped to generate their parent node, given the specified R-tree fanout. After all the parent nodes

are generated and output to another text file, we obtain a new level of the R-tree. Then the new level R-tree is processed as input, and generates another upper level. The packing will continue until only one parent node, the root, is returned.

3.4.2.3 GTR-tree Queries

The parallel query submitted by the end user employs a dynamically generated *JDD(Job Description Document)* string [48] which is parsed and sent to the remote grid nodes, in order to invoke the GTR-tree query procedures on the remote machine.

The GTR-tree query is similar to traversing a B tree, and other tree structures. Given the query rectangle, and the physical file name and node ID for the root node, we perform a stack-based depth-first search, producing the leaf nodes that intersect with the query rectangle. The resulting gains in performance are due not only to the efficiency of tree based search, but also to the parallel nature of the implementation.

Table 3.1: Experimental Data Grid Node Characteristics

Node Name	Location	OS	Processor	Number of CPU	Memory	Java Version
Node-1	Univ. of Mississippi	Mac OS X 10.5.8	Dual-Core Intel Xeon	2	2G	SE 1.5.0_22
Node-2	Univ. of Mississippi	Mac OS X 10.5.8	PowerPC G5	2	2G	SE 1.5.0_22
Node-3	Univ. of Mississippi	Mac OS X 10.5.8	Quad-Core Intel Xeon	2	8G	SE 1.5.0_22
Node-4	Univ. of New Hampshire	Linux 2.6.18	Intel Xeon 5150	4	4G	SE 1.6.0_18

3.5 Experiments And Results Analysis

3.5.1 Test Environment

In order to test the performance of the GTR-tree, a four-node data grid system was constructed using Globus Toolkit 4.2.1. Three nodes were on the same local network at the University of Mississippi, while the remaining node was located at the University of New Hampshire, incurring roughly 50ms RTT latency. Detailed information about the testbed is presented in table 3.1.

Table 3.2: Representative Spatial Queries

	3D Query MBR (String)
Q1	“1400 1400 1400 1400 1400 1400”
Q2	“1500 1500 1600 1510 1520 1640”
Q3	“50 50 50 100 100 100”
Q4	“100 100 100 200 200 200”
Q5	“200 200 200 400 400 400”
Q6	“800 900 1000 1000 1100 1400”
Q7	“1024 1024 1024 1324 1324 1324”
Q8	“1600 1600 1600 2048 2048 2048”
Q9	“900 900 1000 1400 1100 1400”

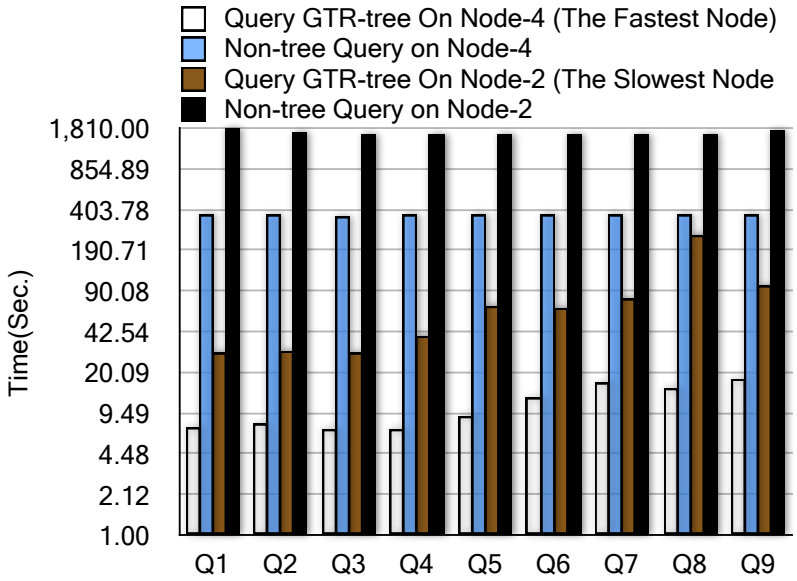
We used the Java API of the Globus Toolkit, the *SQLite3* database, and the *sqlite3odbc* driver as the backend of the RLS. Globus components were used with default settings, unless otherwise noted.

After loading one Million pieces of spatial metadata into the RLS on each grid node, the nine representative spatial queries shown in table 3.2 were used to test the GTR-tree. The entire distributed GTR-tree in the experimental grid is presented in figure 3.6, with fanout 10. On each grid node, the total size of the metadata is around 132MB for one million 3D spatial replicas, each randomly generated with extents ranging from 100 to 512, all within a dataset volume of size $2048 \times 2048 \times 2048$.

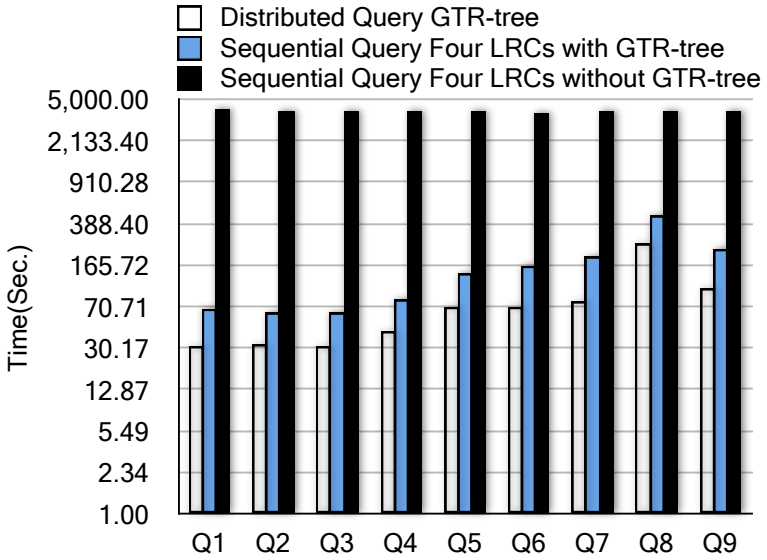
Table 3.3: Number of Replicas Returned & Average Speedups

------	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Average Speedups [1]
Node-1	7	4	178	645	2190	4700	6573	18194	8737	48.9
Node-2	0	3	135	1039	4272	3941	5434	35152	8104	34.4
Node-3	3	34	246	1350	4475	4007	7047	39411	9476	27.2
Node-4	1	24	24	88	615	2045	4091	4503	4954	37.3

[1] Speedup for an individual query is calculated using the ratio of the performance of the GTR-tree query to that of Non-tree query, as shown in figure 3.7(a). The average speedup is obtained by averaging all nine query speedups on each node.



(a) GTR-tree Query Performance compared with plain RLS performance on Node-2 and Node-4, the slowest and fastest processors respectively. Tree fanout was 10.



(b) Distributed GTR-tree Query Performance compared with sequential performance. We show results both with and without the GTR-tree for the sequential cases, in which four LRCs are examined one after another. Tree fanout was 10.

Figure 3.7: GTR-tree Performance. (Using Logarithmic Scale)

3.5.2 Test Results

All results were collected as an average of three runs to reduce the effect of filesystem caching, network behavior, and other variables. Figures 3.7(a) presents speed-up on Node-2 (the slowest node) and Node-4 (the fastest node) respectively with R-tree fanout equal to 10.

While evaluating and fine tuning our query implementation, we made two observations of interest. First, it is best to minimize the length of the stack by only pushing child nodes onto the stack that intersect with the query rectangle, rather than performing an intersection test after popping children off the stack. This shorter stack enhances performance through better memory locality. Second, it is best to retrieve child attributes from the LRC in bulk, rather than individually, due to latency costs. For this reason, we issue a single bulk request to the LRC to retrieve the attributes for up to M tree nodes. We then test each tree node for intersection with the query rectangle, and push only those that intersect. We found this method to be several times faster than the alternatives. The results presented in this chapter were produced with M set to 500.

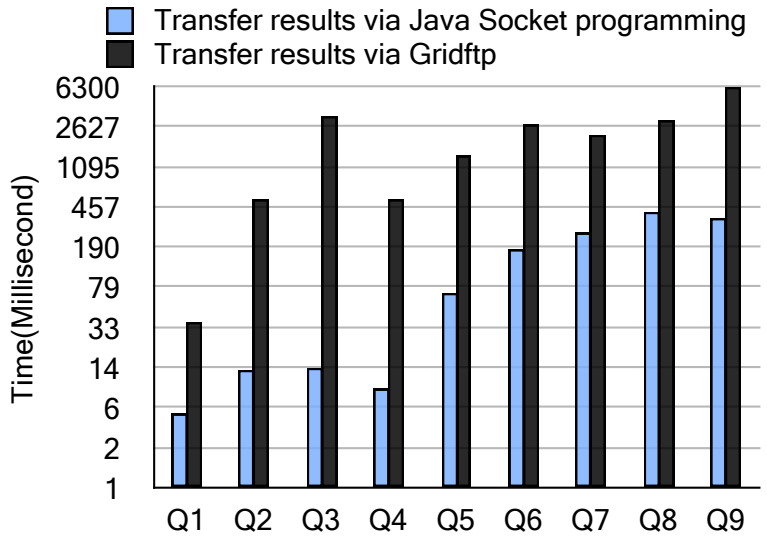


Figure 3.8: Time for returning query results from Node-4 to Node-1 using a Java socket compared with using gsiftp. Queries are sorted by the number of replicas returned in ascending order, as shown in table 3.2. (Using Logarithmic Scale)

Table 3.3 presents the number of replicas returned for each of the queries that are shown

in table 3.2, and the average speedup on each node. Figure 3.7(b) compares parallel *R tree* performance with serial performance both with and without the *R tree*, where speedup reflects the contribution of both the R-Tree and parallel execution. In this case, the query results are left on the remote grid node for further processing. In the serial cases, the query rectangle was presented to each node sequentially and the total elapsed time recorded. In the non-tree cases, replica selection was performed by comparing each MBR in the database with the query rectangle.

We investigated both storage and temporal overhead. For one million replicas, the storage overhead of the GTR-tree was found to be 11% for a fanout of 10, and around 5% for a fanout of 20. The average overhead of GRAM is 5.8 seconds for each query when using the Java API. GRAM overhead includes client parsing of the RSL string, authentication, network latency, JVM startup, communication between the JVM and the GRAM server, and job cleanup. We were careful to verify similar behavior for various combinations of operating systems. The time overheads for returning results via gridFTP and via Java sockets are shown in figure 3.8.

3.6 Chapter Summary

We have described the GTR-tree, an R-tree implemented on top of the Globus Toolkit Replica Location Service component and using a relational database to store all the R-tree nodes. Using a tree data structure with large fanout produces the expected gains in performance compared with exhaustive search. More significantly, our implementation enables existing grid infrastructure to address the important problem of identifying intersected spatial replicas in an efficient manner. This is a crucial step for partial replica selection. We reorganized the metadata into an R-tree structure stored in a relational database.

The current implementation provides many avenues for future research. Robustness in the event of node failure could be addressed by duplicating tree information on multiple nodes. Two additional issues are of particular interest to us. First, the current work assumes that the set of replicas is static. We must extend our scope to also handle insertion, updating the distributed tree as new replicas are created. This problem is not trivial, and introduces questions about packing

algorithms, tree balancing and efficiency, but a significant body of research on this topic already exists. The second issue is the extension of our current algorithm to select from multiple replicas that represent a given region of the data space. Replicas may differ not only in physical location, which affects network costs, but also in the manner they are stored, and the devices they are stored on. Estimates of both network and storage costs can be used to select the replica that will yield best performance. The existing GTR tree efficiently identifies the set of replicas that intersect the query rectangle, but the job of choosing optimally from among this set is described in chapter 5.

CHAPTER 4

PARTIAL REPLICA LOCATION SERVICE WITH THE MORTONIZED AGGREGATED QUERY R-TREE

In this chapter, we first analyze our GTR-tree implementation presented in previous chapter. Then, in section 4.3 the MAQR-tree implementation is presented. In section 4.4, the advantages and usefulness of proposed techniques are validated through experiments. We conclude by summarizing this chapter and pointing out future research directions in section 4.5.

4.1 Limitations of the GTR-Tree

Our approach described in previous chapter was implemented on top of the Globus Toolkit RLS. In particular, we repurposed the *Physical File Name(PFN)* to be a reference to a tree node. We then associated two string attributes "MBR" and "Info" with that PFN in order to represent its spatial extent and references to its children respectively. To enable the distributed query in a grid, we used the GRAM API to invoke the R-tree traversal routine which was located on the same machine as the RLS server. Query results can be returned by using GridFTP or a Java socket.

The approach described above is still subject to some disadvantages. First, fanout of the GTR-tree is limited by the size of the string attribute in the RLS backend database. Although R-trees of large fanout can be represented in the database by introducing more user-defined attributes, we propose a more elegant solution in section 4.3.2.2.

Second, GTR-tree node representation was not optimized and was constrained by the design of the existing table structure in the backend database. In particular, one tree node was stored in three different tables, which made the query and updates inefficient due to an expensive join operation.

Third, although the GTR-tree dramatically prunes the search space, some amount of metadata was still sent from the server code to the query routine via local socket, which incurs some overhead. Also, because we needed to use existing Globus tools, we incurred additional latencies associated with *GRAM* and *GridFTP* when performing a distributed query.

4.2 A New Approach

In this chapter, we modified the *Globus Toolkit RLS* to support spatial metadata management in a grid. We evaluated the performance of an R-tree stored in a relational database, and some related factors which influence the performance of spatial queries or updates, including *Fanout*, *Morton Space-filling Curve* and *Tree Node Representation*. Our proposed *RTP* technique further improves the query performance.

4.3 RLS in Globus Toolkit

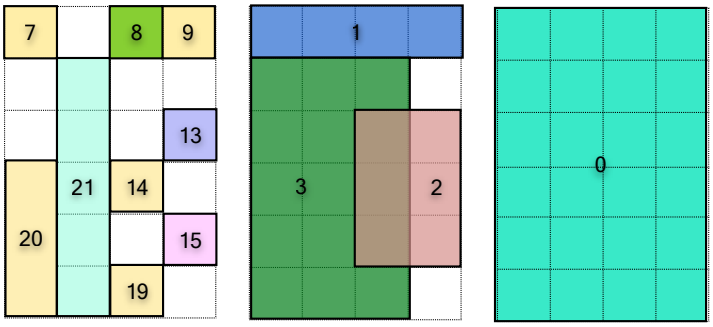
Communication between client and server in the Globus RLS uses a simple string-based *RPC* protocol [21][50], which relies on the *Grid Security Infrastructure(GSI)* and the globus_io socket layer from the Globus Toolkit [19]. Method names, parameters and results are all encoded as null terminated strings.

An *RPC* method invocation in the Globus RLS includes several steps. First, the client sends the method name and all the arguments to the server. A thread on the server searches through a method array for an element matching the requested method name. After the matching method is invoked, execution results are sent back to the client. It follows that we can easily extend the functionality of the RLS by adding new entries to the method array, where each entry includes a method name and a reference to the code itself.

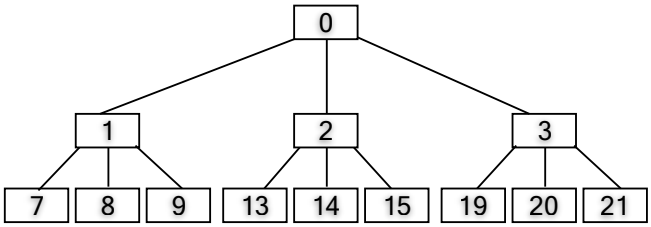
4.3.1 Modifying RLS

The current work extends the *RLS* in Globus Toolkit 5.0.3 for spatial replica selection in several ways. First, we added a new table to the backend database, allowing all tree nodes to be stored in a single database table. Second, we added spatial replica query and insertion methods to

the *RPC* methods array on the server, to service the spatial requests from the clients. Third, we implemented new methods on the server to communicate with its backend database, where query aggregation is used. Fourth, we added the spatial replica query and insertion functionality to the RLS client tool, client *C API* and *Java API* of the existing RLS.



(a) Layout of 2D *spatial* replicas same as figure 3.5.



(b) R-tree with fanout = 6 and out degree = 3 for figure (a).

Node_id (int)	Info (varchar)
0	“<maqr><numChild>3</numChild><MBR>0 0 4 6</MBR></maqr>”
1	“<maqr><numChild>3</numChild><MBR>0 0 4 1</MBR></maqr>”
……	……
21	“<maqr> <numChild>0</numChild><MBR>1 1 2 6</MBR> <address>gsiftp://node1.olemiss.edu/data/d8.bin</address> </maqr>”

(c) MAQR-Tree node representations for figure (b).

Figure 4.1: An MAQR-tree example on a single grid node. The number in each tree node indicates the node id.

4.3.2 The MAQR-tree

To distinguish from the previous GTR-tree implementation of an SRLS, we call our new work the *MAQR-tree*. Two critical factors influence tree performance. First, we consider how to select sets of MBRs to be siblings of a single parent, sometimes called *clustering* [43]. Ideally, sibling nodes would be nearby in space, and also stored close together in the underlying database to improve I/O performance. The *space-filling curve* is a useful tool for ordering child MBRs in a manner that addresses these concerns. The other critical factor is how to efficiently represent the tree structure in the RLS database. Reducing the storage volume for tree representation and improving access speed are both important goals here.

4.3.2.1 Morton R-tree

To construct the Morton R-tree, all spatial replicas are first sorted according to the Morton values of their center. Then we construct the *R-tree* in a bottom-up manner based on the specified *fanout* and the *out degree* of the tree node [49]. Replicas whose MBRs are associated with adjacent Morton values will be clustered into the same tree node. We evaluate the MAQR-tree performance with and without the Morton re-ordering in section 4.4.

4.3.2.2 New Tree Node Representation

We added a new table named *t_spatial_rep* to the RLS backend database, to store the meta-data of all spatial replicas. Two different table schemas have been designed, and the performances of two schemas are compared in section 4.4. Figure 4.1 describes one representation of the improved GTR-tree. In order to emphasize different tree representations, replicas in figure 4.1 are grouped into tree nodes in a same way as in figure 3.5.

Given a current node id pid, the tree fanout f, and out degree d, we can compute all its

child ids using:

$$first\_child\_id = pid \times f + 1 \tag{4.1a}$$

$$last\_child\_id = first\_child\_id + f - 1 \tag{4.1b}$$

$$last\_occupied\_child\_id = first\_child\_id + d - 1 \tag{4.1c}$$

Also, given the child id *cid*, we can calculate its parent id *pid* by:

$$pid = \left\lfloor \frac{cid - 1}{f} \right\rfloor \tag{4.2}$$

To construct a R-tree with n spatial replicas, the improved GTR-tree node ids on each level can be determined by using equation 4.1 and the tree height H, defined as length of the path from the root to the deepest node in the tree.

$$H = \lceil \log_d(n) \rceil \tag{4.3}$$

During tree construction, note that only d replicas are clustered into the same tree node, the rest of $(f - d)$ unused node ids and all their descendant ids will be reserved for tree updates, with $d <= f$.

4.3.2.3 Advantages of The New Representation

There are several advantages associated with the new tree representation, relative to our previous work and other designs.

- *Node ID Representation*: We used an integer to represent node id. In the underlying database, the performance is improved when tree nodes are retrieved using integer keys, especially after a relational database *Index* has been constructed on the column *Node_id*.

- *Table Simplification*: Metadata of *spatial* replicas is stored in a single table, and no relational join is performed during a query.

- *No explicit representation of child and parent ids.*: The storage size of the tree node is

dramatically decreased, resulting in a more compact backend database, because no stored reference to the children is required. Instead we can calculate all child ids on the fly using equation 4.1.

- *Arbitrarily large fanout values*: The *fanout* of the MAQR-tree is no longer limited by the size of a table column. We can create a MAQR-tree with a fanout of more than one thousand, for instance. Although a large number of node ids are reserved in the tree, the storage utilization is not influenced by the tree fanout.

- *Query Aggregation*: Database queries can be aggregated into a single database transaction using a range of tree node ids.

- *Support for insertion and deletion*: the new MAQR-tree is not a *Packed R-tree*. Instead it is dynamic, and allows insertion and deletion of replicas in the tree.

- *Prefetching*: We have developed a prefetching scheme that hides I/O costs as the tree is traversed.

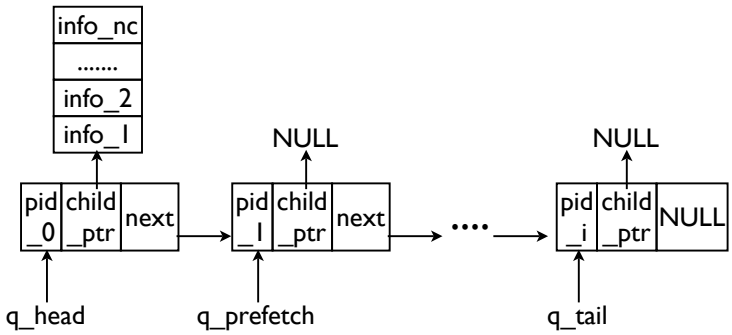


Figure 4.2: The queue structure used for R-tree prefetching with query aggregation.

4.3.3 R-tree Prefetching

In this work, the MAQR-tree is stored in the RLS's backend relational database. First, we used the existing storage infrastructure of the RLS. In addition, the existing RDBMS provides

important functionality that makes it easy to implement. For example, the GTR-tree updates can use the atomic transaction mechanism of the underlying DBMS to maintain consistency when the server is simultaneously processing both queries and updates.

As described in section 6, RTP is based on the concept of application-disclosed access patterns. We traverse the R-tree using breadth-first search, which typically works with a queue structure. During traversal, processing one internal tree node may result in many enqueues of its children. This is especially true when query MBRs and the out degree of the R-tree are larger. These children in the queue can be considered as the prior knowledge of access patterns disclosed by the traversal, which we will certainly access in the future.

4.3.3.1 Query Aggregation

To reduce the latency cost associated with each replica query, in prefetch thread we aggregate a group of replica reads into one larger read. We call this technique *query aggregation*. Under the MAQR-tree representation, the prefetch thread first reads the tree node id pointed to by a prefetch pointer, and next computes the *first_child_id* and the *last_child_id* using equation 4.1. Then the prefetch thread will perform a range query on the table column *Node_id* shown in figure 4.1(c), and all child information is retrieved in one transaction. We constructed a relational index on the table column *Node_id*, so the range query on that column is fast. The next section presents RTP implemented with query aggregation.

4.3.3.2 RTP Implementation

The queue used in RTP is shown in figure 4.2. Each queue element has three fields. The field *pid* stores the tree node id whose MBR intersects with the query bound. The *child_ptr* field points to a chunk of memory storing all child information of the *pid*. The last field points to the next queue element. In addition, the entire queue structure has three pointers, *q_head*, *q_prefetch*, and *q_tail*.

Two threads are used for RTP, a traversal thread and a prefetch thread. When the queue contains at least one prefetched element, the traversal thread will dequeue an element and perform

an intersection test on all children stored in the field *child_ptr*. It then enqueues the child ids whose MBRs intersect with the query MBR. When no prefetched elements are available, the traversal thread sleeps.

The children placed in the queue by the traversal thread will not yet have their information loaded from the database. When such elements exist, the prefetch thread will process the element indicated by the *q_prefetch* pointer, reading the child information into memory from the backend database for this node id. The prefetch thread then sets *child_ptr* to point to this memory and updates *q_prefetch*. When no queue element needs to be prefetched the prefetch thread sleeps.

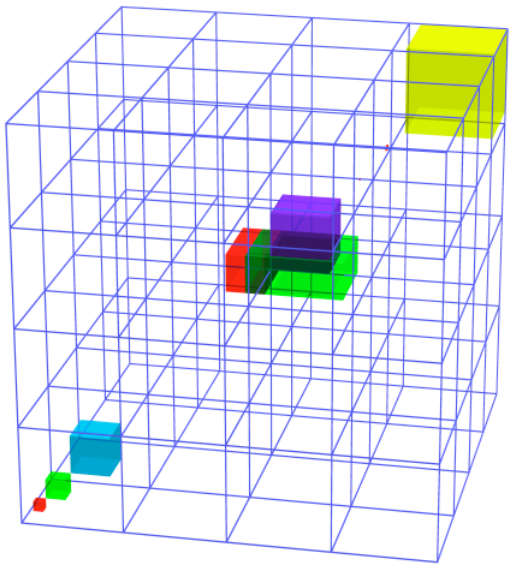


Figure 4.3: The nine representative *spatial* queries used in our tests. Two are extremely small, but are present in the upper right, between the yellow and purple blocks.

Table 4.1: Experimental Grid Node Characteristics

OS	Processor	Cores	Memory	Java version	Globus Toolkit	unix-ODBC	MySQL database	MyODBC library
Linux 2.6.18	Intel Xeon 2.40GHz	16	24G	SE 1.6.0_23	v5.0.3	v2.2.11	v5.0.77	v3.51.26

Table 4.2: Number of Replicas Intersected With Query in Grid

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Average	1995	2578	1288	3405	10667	15990	20181	22351	26523
Maximum	2063	2667	1350	3534	10865	16221	20484	22493	26822
Minimum	1911	2518	1243	3307	10527	15786	19976	22193	26347

4.4 Evaluation

The *Distributed Research Testbed (DiRT)* is a multi-site instrument for developing and evaluating the performance of distributed software. We constructed a twenty three-node grid on *DiRT* using a version of Globus Toolkit 5.0.3 modified to include our spatial replica location service. In the grid, eight nodes are located at the University of Mississippi, seven of them at the University of Florida, five nodes at the University of Chicago and three nodes at the University of Hawaii. The characteristics of each grid node are described in table 4.1. We use unixODBC manager and MyODBC driver to communicate with MySQL server.

We used independent datasets on each grid node. One million 3D replicas were randomly generated on each grid node. One object randomly generates the center point of the 3D rectangle in the entire dataset domain (a 2048^3 cube). Another Random object randomly generates the length of replicas along each axis, which is bounded in the range of 1 to P. The following tests use $P = 512$, unless otherwise noted.

The total size of the metadata for one million replicas is 120M. The average size of metadata for one replica takes 115 bytes, and the physical address string for each replica takes an average of 54 bytes. In the test, the physical address and the id of replicas are transferred back to the clients.

The same nine representative spatial queries used in [51] were also used for testing in this chapter. Queries include one point query and eight rectangular queries of various sizes. The extent of these nine spatial queries, relative to the entire dataset (a 2048^3 cube) are shown in table 3.2 and visualized in figure 4.3. For each query, table 4.2 shows the average number (per grid node) of replicas that intersect that query, and the maximal and minimal number of replicas returned in

the grid. All the following results were collected as an average of five runs to reduce the effect of filesystem caching, network behavior and other variables.

We performed experiments under two file caching conditions: a *warm cache* and a *cold cache*. A warm cache is defined as the state in which the file system has already cached the metadata that we are retrieving. The same query repeated more than once may be satisfied from the cache rather than from disk. For the cold cache case, we unmounted and remounted the file system where the RLS's backend database is stored, which discards file system cache contents. Also, we flushed all caches inside the MySQL server and restarted the SRLS server for each query.

When we tested performance of various MAQR-tree *fanout* and *out degree* values, we found that a MAQR-tree with out degree of 50 yields best performance on average. In the following tests, we used the MAQR-tree with out degree of 50, and fanout of 70.

Column Name:	id	x_min	y_min	x_max	y_max	address
Type:	int	int	int	int	int	varchar

Figure 4.4: The schema used for the pure relational implementation, components of the MBR are stored as integers.

```
select address from t_spatial_rep
where (not ((x_min > qxmin and x_min > qxmax) or x_max < qxmin)
    and not ((y_min > qymin  and y_min > qymax ) or y_max < qymin));
```

Figure 4.5: SQL statement for conducting spatial queries without a MAQR-tree. Used with the schema shown in figure 4.4, this SQL statement can retrieve replicas that intersect with a bounding rectangle $\{qx_{min}, qy_{min}, qx_{max}, qy_{max}\}$. Performance was found to be worse than the MAQR-Tree on average.

Table 4.3: Advantage of MAQR-tree Query Over Pure Relational

	Data1	Data2	Data3	Data4
Total Number	1000000	1000000	1000000	1000000
Average number returned	11645	4649	1816	2642
Average speedup	1.9	4.36	7.75	8.13

4.4.1 Experimental Verification

We performed a series of experiments to verify the usefulness of the MAQR-tree and the effectiveness of the tree representation in the database.

We verified that an R-tree in a relational database is useful for spatial metadata by comparing MAQR-tree performance with a non-tree implementation that relies heavily on the underlying relational database. For this pure relational implementation, we used the database schema shown in figure 4.4. Here, MBR coordinates are stored as separate integer fields in the database, making them directly available in SQL queries. Figure 4.5 shows an SQL implementation of a spatial intersection query using this schema. Indexes were built for all fields.

To carefully examine the advantage of the MAQR-tree over the pure relational implementation, we performed the comparison on four different datasets. We generate these datasets randomly as described before, but with different values for P, which determines the maximum size of replicas. This allows us to control the average number of replicas that intersect the query region in the different datasets. The advantage of the MAQR-tree query over pure relational query is presented in table 4.3.

In table 4.3, we describes the average number of replicas returned per representative query on each dataset. And the average speedup is calculated by averaging the ratios of MAQR-tree query performance to that of pure relational for each query. We observed that the MAQR-tree query can be up to 8.13 times quicker than the pure relational implementation.

The reason for the advantages is very likely the multi-dimensional nature of the MAQR-tree. For pure relational implementation, even if an index can be used to quickly find the set of replicas with appropriate x values, a y index can not help with pruning this set further because it contains *all* the replicas in the database. A time consuming exhaustive search is therefore necessary. In contrast, the MAQR-tree query is able to drastically prune the set of replicas under consideration as it encounters multidimensional MBRs at each level of the tree, improving performance enormously.

Table 4.3 also shows that the MAQR-tree has the greatest advantage when the number of

replicas returned is smaller. These more selective queries exhibit more aggressive pruning while traversing the tree, which enhances performance.

Node_id (int)	num_child (int)	x_min (int)	y_min (int)	x_max (int)	y_max (int)	address (varchar)
0	3	0	0	4	6	null
……	……	……	……	……	……	……
21	0	1	1	2	6	"gsiftp://node1..../d8.bin"

Figure 4.6: Alternative representation of the MAQR-tree for the tree in figure 4.1. Components of MBRs are stored as integers.

We also tested the insertion cost with the two implementations. The pure relational implementation relies entirely on the underlying database to support insertion. For the MAQR-tree, we must perform several steps in order to insert a replica. Once the proper tree location has been determined, all siblings (sharing the same parent) after this location must increment its node id by one in order to make space for the insertion. In the rare case that no space is available, a node split operation must be performed [42], taking roughly 100 milliseconds in our tests. Next, the parent and ancestor MBRs are updated to account for the new node.

We tested MAQR-tree insertions with various groups of replica MBRs, including cubic MBRs of 50^3, 100^3, 200^3, and 300^3. We inserted these cube replicas into different regions of the entire domain. It takes an average of 30 milliseconds to insert one replica into a MAQR-tree. The pure relational approach is slightly quicker, taking an average of 18 milliseconds to insert, which is 1.67 times faster than the MAQR-tree. Overall, the MAQR-tree approach is more suitable for both scientific database and replica selection applications, which involves intensive queries and relatively less updates.

We evaluated the choice of table schemas used to represent MAQR-trees. We compared the performance of the schema shown in figure 4.6, with the schema presented in figure 4.1(c). We observed that MAQR-tree query performance with the former schema outperforms the latter by only 7.1% on average when using the MySQL backend. However, the schema shown in figure 4.6 makes the server code inflexible when dealing with n-dimensional datasets. We have to change

this table schema and the server code for different dimensional abilities. In contrast, the MAQR-tree representation shown in figure 4.1(c) requires no change. Furthermore, our RTP is more effective with the schema shown in figure 4.1(c) than that in figure 4.6, as discussed in section 4.4.3. Therefore, in our experiments we use the MAQR-tree representation shown in figure 4.1(c), unless otherwise noted.

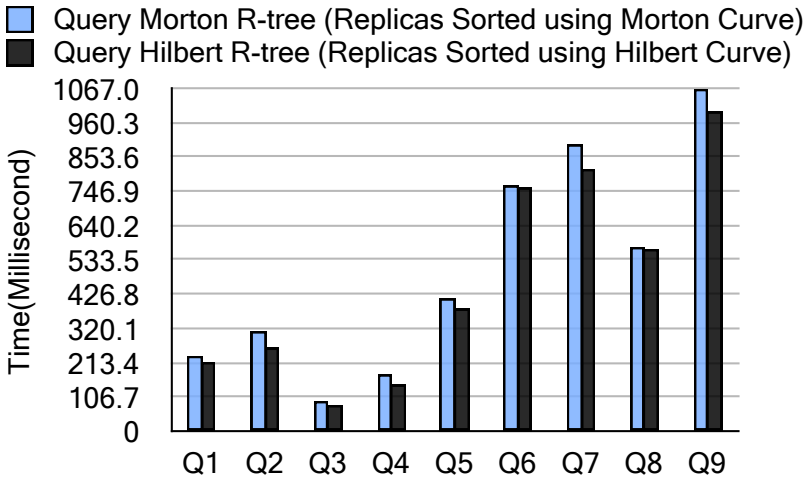


Figure 4.7: Morton R-tree compared with Hilbert R-tree. The Hilbert R-tree performance is 6.7% better than the MAQR-tree query performance on average.

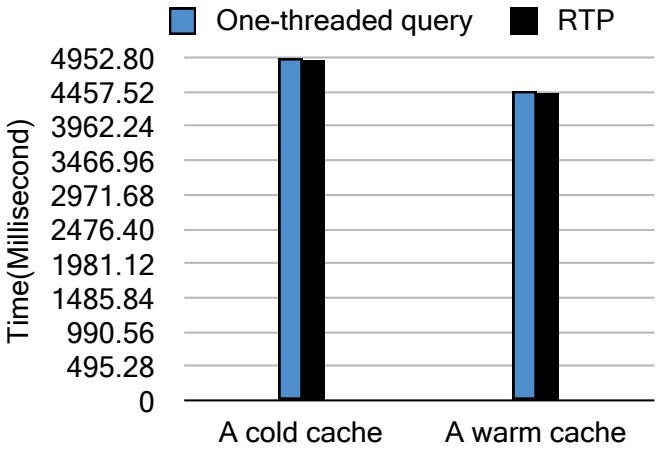


Figure 4.8: Average MAQR-tree query time without query aggregation, in which RTP is less effective, improving the query performance less than 1%.

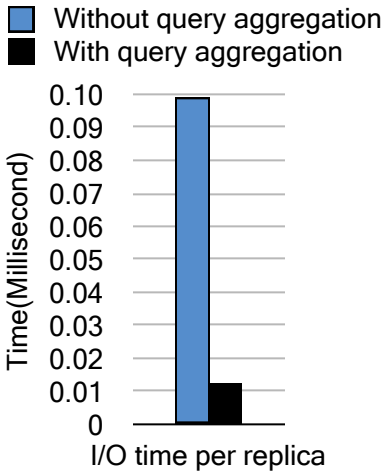


Figure 4.9: Average I/O time per replica in MAQR-tree measured with a cold cache. We find that query aggregation reduces I/O time per replica by 87.4% with a cold cache.

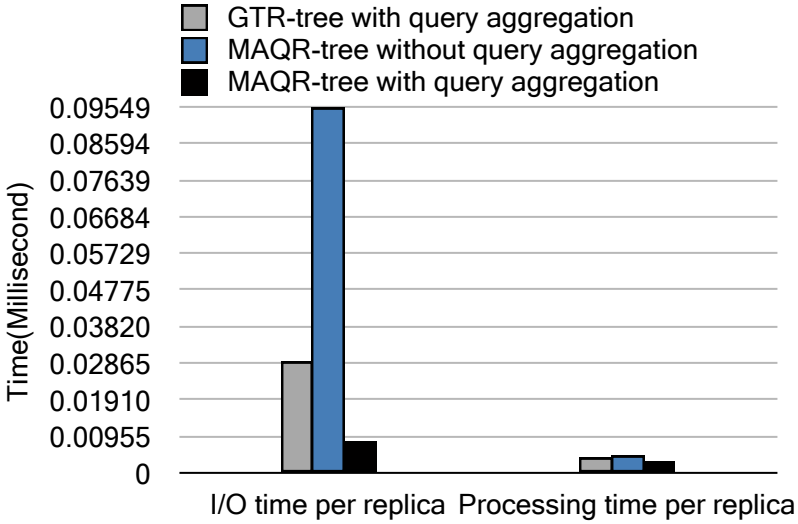


Figure 4.10: Average I/O time and processing time per replica measured with a warm cache. We find that query aggregation reduces I/O time per replica by a factor of 11.2 with a warm cache.

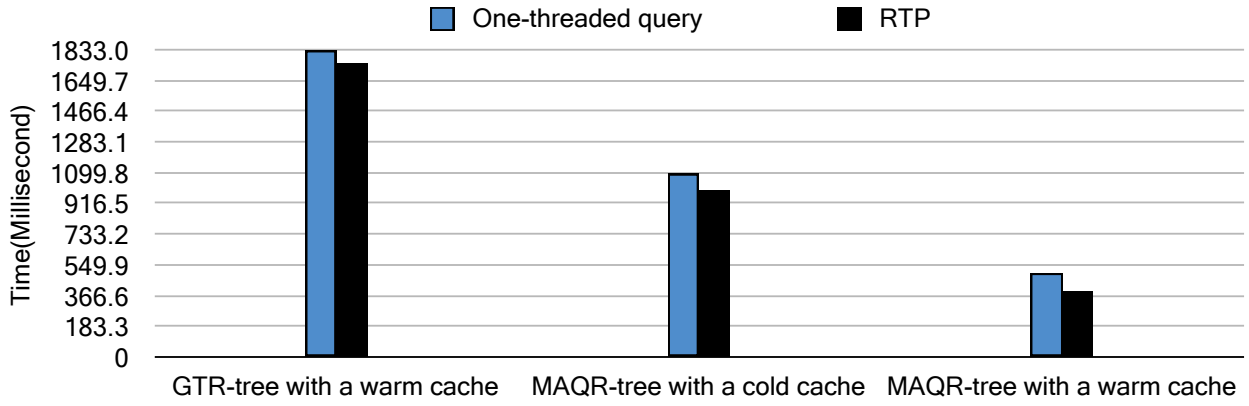


Figure 4.11: Average query time using query aggregation, in which RTP is more effective. RTP further improves MAQR-tree query performance by 9.4% under a cold cache, and by 23% under a warm cache. It also improves the GTR-tree performance by 4% under a warm cache.

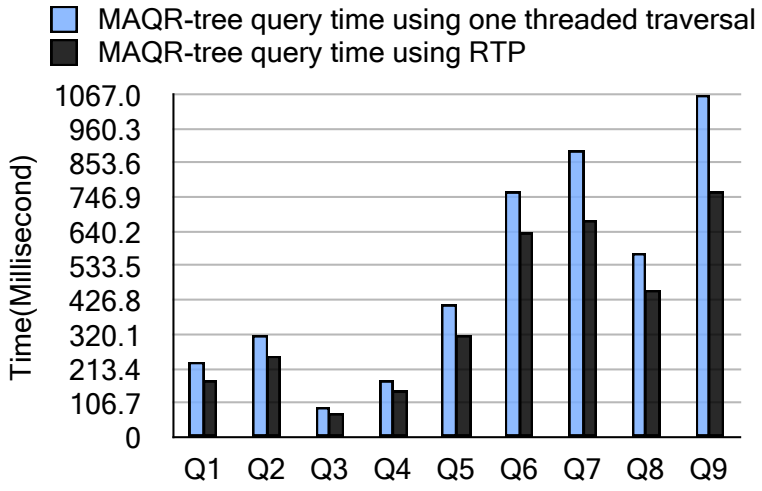


Figure 4.12: RTP compared with single threaded traversal using query aggregation under a warm cache for nine queries. RTP improves performance by 23% on average.

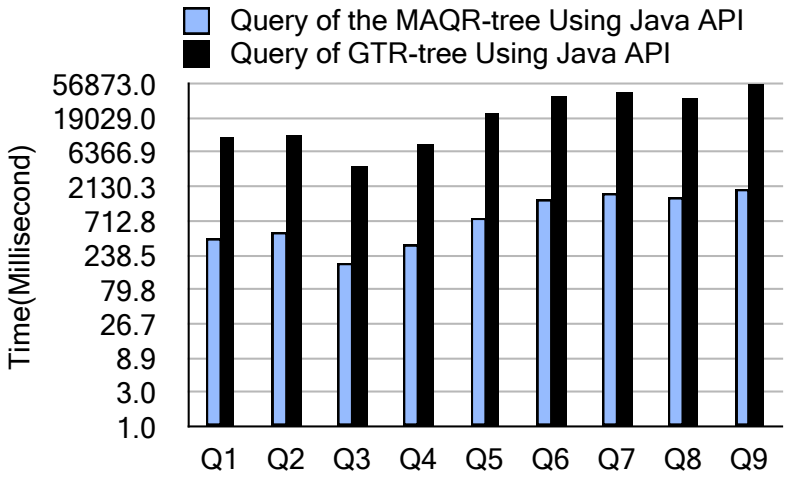


Figure 4.13: MAQR-tree query performance compared with that of GTR-tree. MAQR-tree outperforms GTR-tree 24.5 times on average. (Using Logarithmic Scale)

4.4.2 Morton R-tree

We evaluated the effect of introducing the use of the Morton Space-filling curve to the MAQR-tree implementation, as described in section 4.3.2.1, and found that adding this technique improves performance by more than 30 times on average. Preliminary experiments show that the Hilbert R-tree outperforms the Morton R-tree by a further 6.7% on dataset *Data1* in table 4.3, presented in figure 4.7. Although results presented in this chapter use the Morton curve, we expect slightly better query performance in future by the applying Hilbert curve in the R-tree construction. In this test, we used a table-based algorithm to compute the Hilbert values for 3D MBRs, which is described in papers [52][53].

By associating each MBR with a Morton value, spatially adjacent replicas are mapped to similar Morton values. Therefore, they can be grouped into the same or neighboring parent tree nodes. In this way, the Morton value helps the enclosing MBRs to stay local, resulting in reduced "dead space" and possible overlaps in the tree node [42]. In short, the enclosing MBR is more efficiently used, and the constructed R-tree is more compact. This decreases the number of nodes that must be examined for intersection, thereby improving performance.

To further analyze the benefits of the Morton curve, we dumped each level of the MAQR-

tree into a separate text file for later examination. We observed that the enclosing MBRs of internal nodes are more disjoint, with less overlap between neighboring tree nodes after the Morton Space-filling curve is introduced. For example, before using the morton curve it is observed that most MBRs on the second level of the tree fill up the entire dataset, which means they are much less effective in pruning the search during the R-tree query. In contrast, each of these level 2 MBRs covers only a local region of the entire dataset after the morton curve is used.

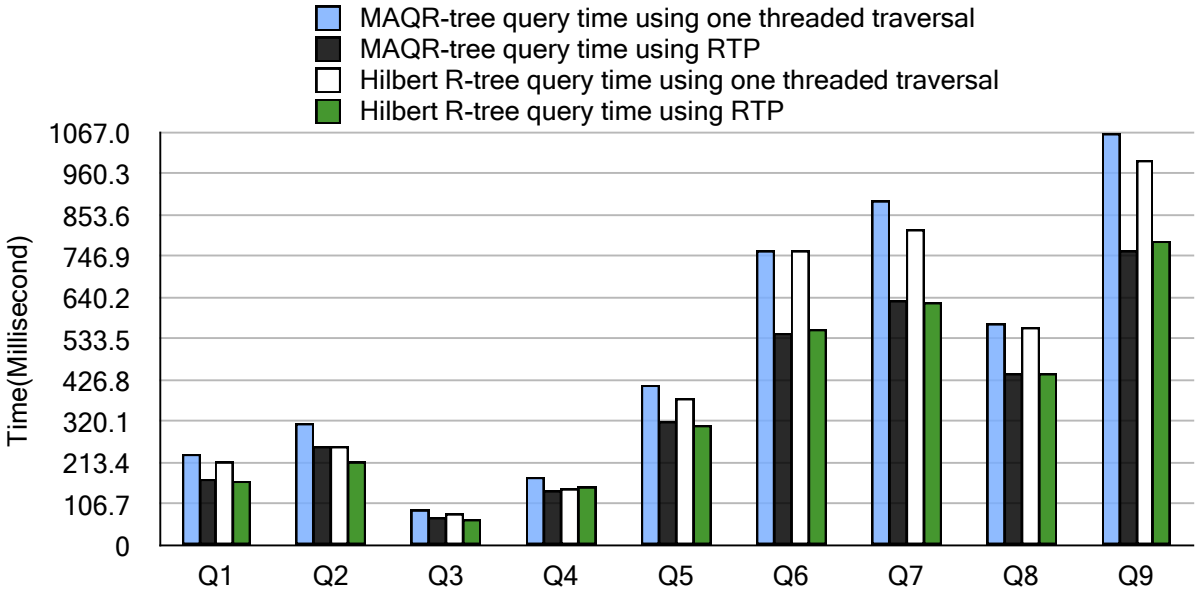


Figure 4.14: Effects of RTP on the query performance of the Hilbert R-tree and MAQR-tree. Without RTP, the Hilbert R-tree is 6.7% quicker than MAQR-tree on average, while with RTP enabled, the Hilbert R-tree and MAQR-Tree have nearly identical average performance.

4.4.3 R-tree Prefetching (RTP)

In order to analyze the behavior of RTP, we tested the average I/O time and processing time per replica under different circumstances. Given a tree node, the processing time per replica includes the time cost of the intersection test on the node's MBR, and enqueueing its children. The I/O time per replica is defined as the time cost to read into memory the information associated with each MAQR-tree node, such as the MBR and replica's physical address. This I/O time consists of two components. The *hardware costs* include disk latency, bus transfer costs, and the like.

Software costs include costs associated with function calls or kernel routines.

It is noteworthy that in this section we only compared the *representations* of the MAQR-tree and the GTR-tree, in terms of I/O and processing time per replica. We tested the efficiency of both the GTR-tree and MAQR-tree representations using the modified Globus RLS code from the MAQR-tree. In section 4.4.4, we compare the MAQR-tree with the entire GTR-tree implementation that is built upon unmodified Globus.

4.4.3.1 Query Aggregation

Our initial implementation of RTP did not use query aggregation. Unfortunately, we found that RTP alone was less effective with either a cold or warm cache, as shown in figure 4.8. Here, the overhead associated with the multithreaded implementation overwhelms any potential benefit. We addressed this shortcoming with *query aggregation*, which amortizes query costs over a collection of what would otherwise be separate queries. Comparing MAQR-tree single threaded query performance with and without this feature activated demonstrates an average 8x performance improvement for Query Aggregation. Figure 4.11 shows that this strategy was effective, and allows RTP to demonstrate significant performance improvements. We use query aggregation with RTP in all following experiments, except where otherwise noted.

Query aggregation is far more effective in reducing I/O time per replica with a warm cache than with a cold cache, which is perhaps counterintuitive. However, our results show that it reduces I/O time per replica by 87.4% with a cold cache and by a stunning *factor* of 11.2 with a warm cache, as shown in figure 4.9 and figure 4.10. With a cold cache, disk access is the dominant component of the I/O cost, while the software cost is minor. When collecting cold cache results we noticed that the first query always took far longer than the next several queries, due to the necessity of accessing the physical disk. This pattern repeats itself over time, but becomes progressively fainter due to the warming of the cache and prefetching by the filesystem. Query aggregation does not have significant effect on the hardware behavior, perhaps because a single disk access retrieves the data required for 50 queries even without aggregation. In contrast, with the warm cache case,

software costs are the dominant component of I/O cost, and it is here that query aggregation shows substantial advantage because it pays these costs far less frequently.

4.4.3.2 RTP Analysis

RTP demonstrates the largest performance gain percentage when used with MAQR-tree under a warm cache, as shown in figure 4.11. The advantage of RTP over a single-threaded method in this environment is presented in figure 4.12. The query performance is improved on average by 23% for nine queries, after using RTP. The largest performance gain is 28% for query Q9 in which more than 26,400 intersecting replicas are returned.

We have already observed that RTP is less effective without query aggregation, as shown in figure 4.8. In our tests we used the same synchronization model regardless of whether query aggregation was used. However, query aggregation significantly coarsens the granularity of synchronized queue access, allowing all children of a node that intersect the query MBR to be enqueued at once. Dequeueing behaves similarly. The net effect is to reduce the number of times queue operations are performed by a factor of *out degree*, which we have set to 50 in our experiments. Because queue access is guarded by a mutex, this reduces the impact of synchronization costs, an important component of the processing time.

When using two separate threads for I/O and computation, the best performance is obtained when I/O and computation costs per element are exactly equal, yielding twice the performance of the single threaded implementation. For this reason, RTP shows more performance improvement when the I/O time and the processing time per replica are well matched. Figure 4.10 shows that computation and I/O costs are best matched in the MAQR tree implementation using query aggregation on a warm cache, and we found RTP performance was improved by 23% over the single threaded implementation. In contrast, for the GTR-tree representation the I/O time per replica is 7.2 times larger than the processing time per replica, resulting in an average of only 4% performance improvement after using RTP. This also accounts for why RTP with MAQR-tree representation in figure 4.1(c) outperforms RTP with representation in figure 4.6 by 17%.

In summary, RTP and query aggregation improve performance significantly, either with a cold cache or with a warm cache. Neither of these two testing scenarios are a perfect model of the real world, but two points should be made. First, cold caches are rare in practice because the filesystem cache will retain data until it has a reason to discard it, meaning that replica data could easily persist in filesystem memory for days. Second, scientific users often access the same "hot-spot" dataset regions repeatedly. In real world cases where the same (or nearby) data is repeatedly accessed, our warm cache results should prove an accurate indicator of performance.

4.4.3.3 Effects of Space-filling Curves

We investigated the influence of RTP on the choice of space-filling curves. As described in section 4.4.2, without RTP enabled, the Hilbert R-tree outperforms the MAQR-tree (which uses Morton curve) in query performance by 6.7%. The Hilbert curve produces more compact nodes, which produces a performance increase during tree traversal because fewer nodes are visited. However, figure 4.14 shows that with RTP enabled, the advantage of the Hilbert curve disappears.

We explain this surprising fact by considering the filesystem cache as an important third factor. This cache views files as one dimensional, so when the application access pattern proceeds through the 1D file without jumping around, the filesystem cache's speculative prefetching is effective. Because the Hilbert curve jumps around in the 1D file space less frequently than the Morton curve, it usually has better performance. When RTP is added, however, our own SRLS is *explicitly* prefetching groups of required nodes, even when those groups span a gap in the 1D file. This makes the filesystem cache's speculative prefetching less useful, which in turn removes the Hilbert curve's advantage.

4.4.4 Comparison with GTR-tree Implementation

To make a fair comparison of the MAQR-tree with our previous GTR-tree, we tested on a single grid node using a same dataset. We compared the efficiency of the tree representation and associated techniques by sorting all replicas for both MAQR and old GTR-tree with the same out degree. We observed the performance of the MAQR-tree is on average 24.5 times better than the

previous GTR-tree, as shown in figure 4.13. We used the Java API for these tests, and excluded the time cost for establishing a secure connection. Results are shown on a logarithmic scale.

We attribute these speedups to the advantages of the MAQR-tree over the GTR-tree described in section 4.3.2.3 and section 4.1. We observed that the I/O time cost per replica for MAQR-tree representation is more than 20 times quicker than that of GTR-tree mainly because previous GTR-tree implementation did not apply query aggregation. In MAQR-tree, we also eliminated the local socket network communication between the server and the spatial query routine. These optimizations easily accounts for the improvements shown in the experimental results.

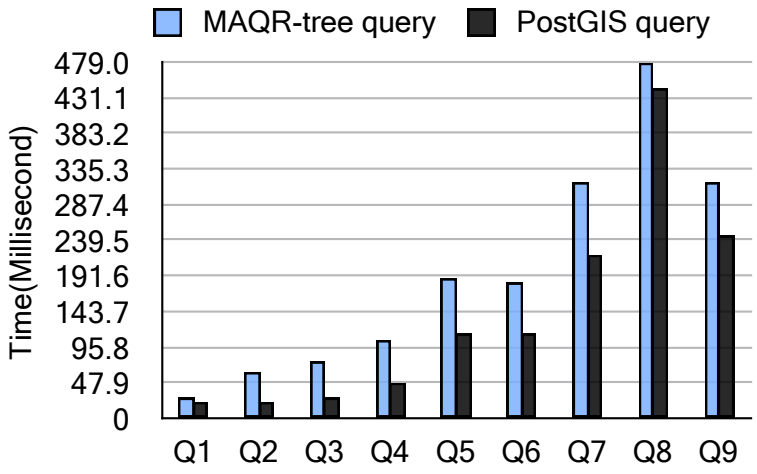


Figure 4.15: MAQR-tree query performance compared with that of PostGIS for 2D dataset. The PostGIS using GiST indexing outperforms our proposed MAQR-tree with RTP by 28.5% on average.

Table 4.4: Number of 2D Replicas Intersected with Each Query In PostGIS Test

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Number of 2D replicas returned	68	241	1019	3303	11066	11163	23715	49581	26558

4.4.5 Comparison with PostGIS

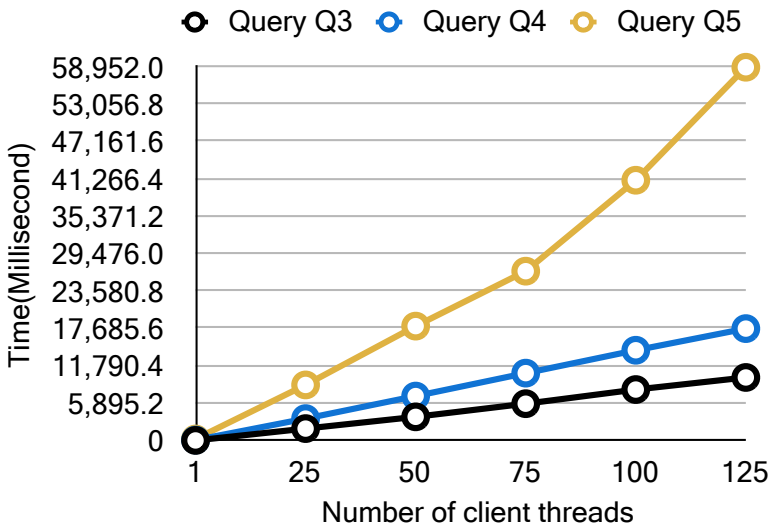
We compared our SRLS with *PostGIS 1.5.3*, one of the leading extensions to relational databases targeted toward *Geographic Information Systems (GIS)*. We randomly generated one

million 2D replicas and compared query performance, shown in figure 4.15. The number of replicas intersected by each query is presented in table 4.4. The SRLS uses the MAQR-tree with RTP, while PostGIS uses a built-in GiST index [54]. PostGIS outperforms our MAQR-tree implementation with RTP by 28.5% on average. Although PostGIS provides an intersection test for the 2D MBRs commonly found in GIS, higher dimensionalities are not yet supported [55]. Our own SRLS supports MBR intersection tests for three or more dimensions, making it more suitable for general scientific use.

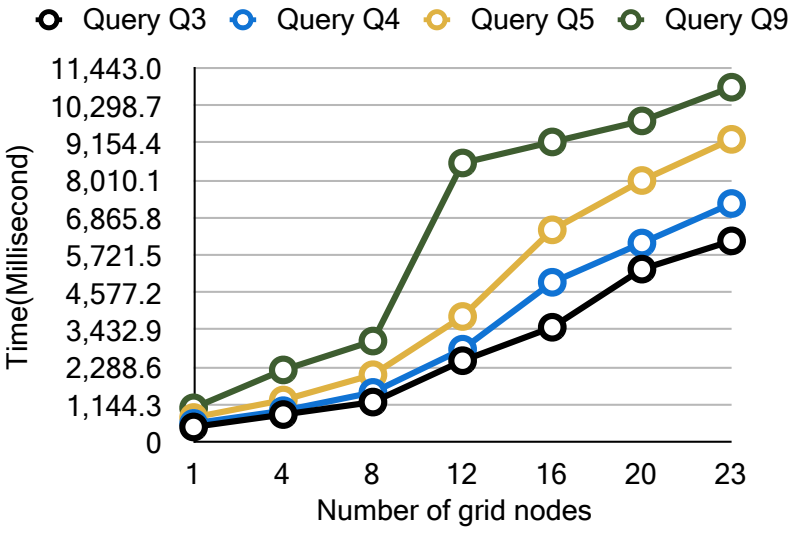
4.4.6 Multi-client and Distributed Experiments

We wrote a multi-threaded client program using our C API and the *pthreads* library. We conducted the multi-client tests with the server and clients on a same machine, on which our threaded client program allows us to specify how many client threads are created simultaneously. In the distributed query, initially we submitted queries on a client grid node at the University of Mississippi to a server node at the University of Chicago. We then increased the number of server nodes by adding a combination of at least two remote nodes and one local node.

The results of the multi-client and distributed query are shown in figure 4.16. Figure 4.16(a) shows that server performance scales well as the number of simultaneous client threads increases. In (b) we see that performance also scales well as we increase the number of server nodes and total number of replicas. Each server maintains one million replicas. With twenty three nodes we are handling twenty three million replicas, but execution time is less than ten times as long as for one million replicas. We observed a sharp increase in the query time for Q9 using between eight and twelve nodes in (b). This is due to adding the server nodes at the University of Hawaii and the larger amount of replicas returned, which are transferred via a network with latency more than twice higher. The average number of replicas intersected with each query on each grid node is shown in table 4.2.



(a) Multi-client Experiment of the spatial replica location service using RTP.



(b) Distributed query using one to twenty three nodes using SRLS with RTP enabled.

Figure 4.16: The multi-client test in (a) shows that server performance scales well as the number of simultaneous client threads increases. In (b) we see that performance scales well as we increase the number of server nodes and total number of replicas. In each test, the total number of replicas is the number of nodes multiplied by one million.

4.5 Chapter Summary

Our eventual goal is a system where both data and computation are truly distributed. We envision applications where only a subvolume of a larger dataset is required for a computation, and where that subvolume may be available from various combinations of different sources. If we hope to provide more than simple batch computation for such applications, we must be able to rapidly identify the set of partial replicas that intersect with the required subvolume, and then choose an optimal combination of replicas to read from. This chapter presents work that rapidly identifies the set of intersecting replicas in a distributed environment.

Our previous chapter solves this problem with an implementation that lay entirely on top of an unmodified Globus Toolkit. This approach has the advantage of easy deployment using existing software infrastructure, but must also pay a performance penalty. This chapter examines the performance benefits of an approach that modifies the Globus source code to provide a more efficient implementation. We also provide important new functionality in the form of insertion and update operations.

With regard to the internal implementation of the R-tree data structure and associated operations, we have also added several improvements. Ordering tree node children according to the Morton curve was very effective, as was Query Aggregation. Taking advantage of prefetching through RTP produced additional 23% improvement. The net result is performance roughly 24.5 times better than our previous work. Experiments show that a single server with one million replicas in the database can scale up to at least 125 client threads, when handling large spatial queries. Our preliminary experiments suggest that performance of the proposed MAQR-tree is comparable with PostGIS query performance.

CHAPTER 5

PARTIAL REPLICA SELECTION

After a set of replicas that intersect a spatial query have been located by the SRLS, it remains to *select* a set of partial replicas that will satisfy the query with good performance. This replica selection problem is the focus of this chapter.

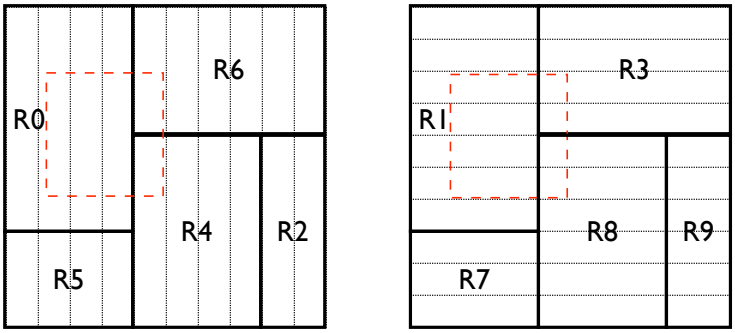
We make two major contributions here. First, we propose a cost model that can accurately predict for which access patterns the operating system will trigger its prefetching mechanism. Operating system prefetching causes the access time cost to vary by a factor of 2 to 3. We use a *storage model*, from which we can derive the disk access pattern for a query. We then estimate the time cost for the query, considering the operating system's caching and prefetching and hard disk behavior.

Second, we describe a greedy algorithm for the partial replica selection problem, which has been shown to be NP-complete [8]. Our partial replica selection algorithm takes into account storage organization, operating system prefetching, hard disk parameters, network performance and load balancing. Experiments show that our replica selection algorithm can choose replicas efficiently and with excellent results. Using the proposed cost model for evaluation, we observed the performance of the solution found using our algorithm is on average always at least 91% and 93.4% of the performance of the optimal solution in 4-node and 8-node tests respectively.

The rest of the chapter is organized as follows. In section 5.1, we describe our storage model, device model and model of replica transfer time. Section 5.2 presents our design and implementation of the replica selection algorithm. We validate our models and algorithm in section 5.3.

5.1 Models

While our work [51, 56] solved the replica *location* problem, here we concentrate on replica *selection*. Assuming that we already have a set of partial spatial replicas that intersect with a given spatial query, we must now choose which replicas to read from in order to achieve best performance. A single area of the domain may be represented by several replicas residing on different hosts, or stored using different file formats. For each replica, we need to accurately estimate the cost of retrieving the subset of the replica that intersects with the query. Such an estimate requires several models, which we present in this chapter.



(a) A 2D dataset split into partial spatial replicas stored in column-wise order on disk.

(b) Here, the same partitioning is used, but replicas are stored in row-wise order on disk.

Figure 5.1: We partition the same dataset into partial spatial replicas in (a) and (b), but using different storage orders. To illustrate the partial replica selection problem, we assume replicas in the same region have a same *Minimal Bounding Rectangle(MBR)*. The dotted rectangle represents a spatial query Q.

5.1.1 Storage Models

In the work [30], the concept of a *storage model* was introduced. A *storage model* is used to map the n-dimensional data space to a 1 dimensional file offset or memory address. A storage model can also be used to compute the number of separate read operations required to satisfy a subblock query. Essentially, a storage model describes how the dataset space relates to the underlying storage device.

Many n-dimensional files are stored in *linear* order on disk, as shown in figure 5.2. The *rod storage model* can be used for such files, where "rod" refers to a sequence of data elements that

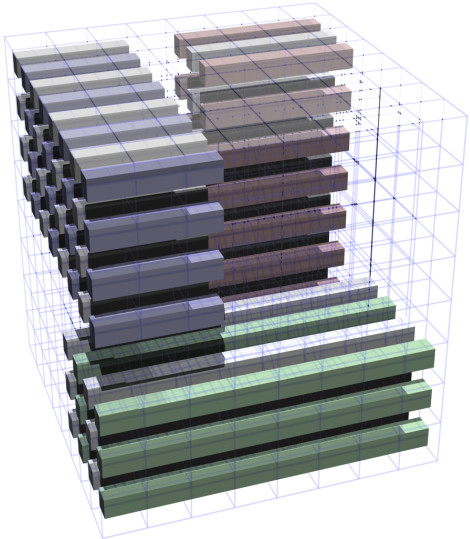


Figure 5.2: For datasets that fit the rod storage model, a subset query can be broken up into a collection of query rods. Three different queries are shown here, using red, green, and blue rods. Rods can then be grouped into slices, indicated by the alternating light and dark colors. In the underlying file, rods within a slice will be relatively close together, while moving from slice to slice requires a much larger jump. The three queries shown have the same shape ($4 \times 3 \times 8$), but their orientation to the rod axis causes the number of query rods and slices to vary considerably between the queries.

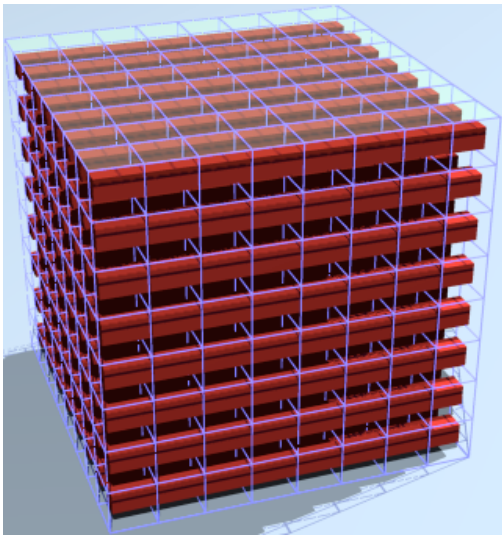


Figure 5.3: A volumetric dataset stored on disk with rod storage model. Each red "bar" is referred to as a space rod in rod storage model.

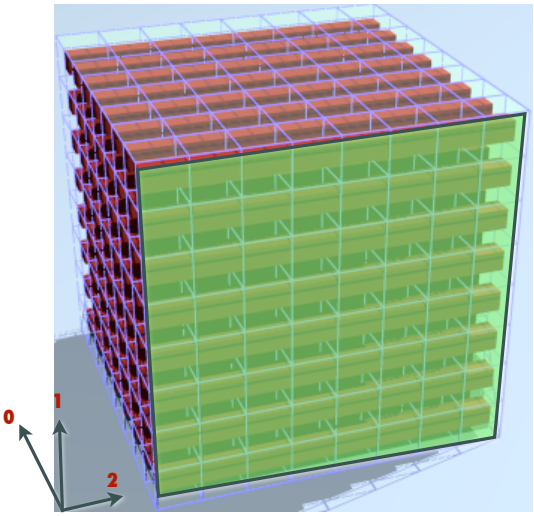
are contiguous both on disk and in the index space [30], which is illustrated in figure 5.3.

Because rods are contiguous on disk, they can be read with a single transaction, so counting the rods contained in a subblock query is a very good indicator of how many disk accesses a query requires. For example, the three queries shown in figure 5.2 have the same dimensions ($8 \times 4 \times 3$), but their orientation to the rod axis causes the number of query rods and slices to vary considerably between the queries. We can use this information to infer that the green query in the lower front of the volume will take the least time to read, since it contains only 12 rods. The others will require 24 and 32 reads. Of course, if the rods of the underlying dataset were oriented differently, either the red or blue query might be the fastest.

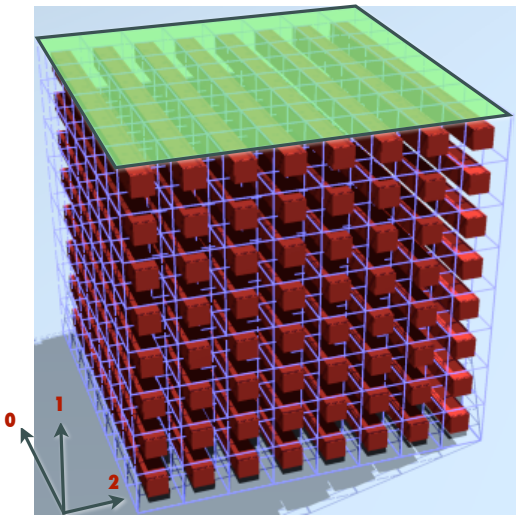
In another example shown in figure 5.1, a 2D dataset is partitioned into five partial replicas in figure 5.1(a), with each replica stored on disk in a column-wise order. In figure 5.1(b), we keep the same partition, but with a row-wise storage order. The red dotted rectangle represents a spatial query Q. In order to answer the query, it needs 3 disk reads from replica $R0$, but requires one more read from its copy of $R1$. Therefore, the storage ordering of a replica has great influence on the performance.

The rod storage model denotes the direction of the rods using an *axis ordering*, which is simply a ranking of axes from outermost to innermost. "Innermost" and "outermost" suggest position in a set of nested *for loops*. Axes are labeled with numbers, so an axis ordering is really just a list of integers.

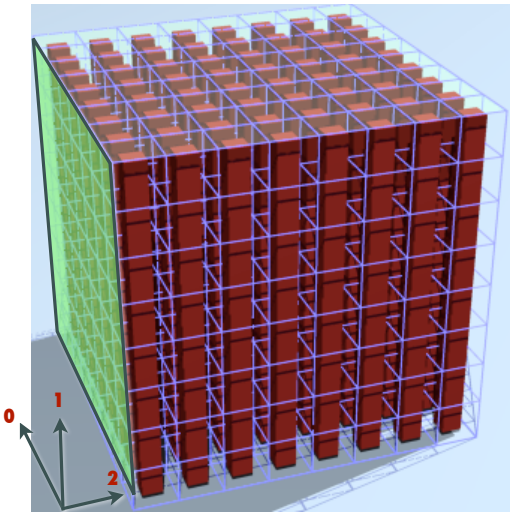
When an axis ordering denotes the order in which data is stored on disk, we call it a *storage ordering*. The innermost axis is known as the *rod axis* (and changes most frequently) while the outermost axis changes least often. For three dimensional data, this outermost axis is known as the *slice axis* because we can think of the volume as composed of a series of two dimensional slices positioned orthogonally along this axis. Each slice is contiguous on disk. In the underlying file, rods within a slice will be relatively close together, while moving from slice to slice requires a much larger jump. For three dimensional files, there are $\binom{3}{2} = 6$ different orderings, but orderings $\{0,2,1\}$, $\{1,0,2\}$, and $\{2,1,0\}$ are representative, since they have different slice and rod axes. We



(a) Storage ordering $\{0,1,2\}$.



(b) Storage ordering $\{1,2,0\}$.



(c) Storage ordering $\{2,0,1\}$.

Figure 5.4: We show three different storage orderings. In figure (a), we first store data along axis 2. When reaching the end of one space rod, we move to the next space rod along axis 1. Axis 2 and axis 1 determine the direction of a *Slice*, as shown in green color, where space rods are grouped into slices. It is similar for orderings in figure (b) and (c).

show three different storage orderings in figure 5.4.

In this work, we employ the rod storage model to map spatial queries to some set of accesses in the underlying one dimensional file. To do so, we need to know the spatial dimensions of the file, especially the *space rod size*, which is the length of the data space along the rod axis. The *query rod size* is similar, but applies only to the query volume rather than the entire data space.

Since an n-dimensional dense array can always be recursively described by one or more $(n-1)$ dimensional subsets, we call a 2-dimensional subset a *Slice*, which is determined by the fastest changing axis and second fastest changing axis in rod storage model. Direction of a slice is shown in figure 5.4 with a green plane. We also define the *space rod size* as the dimension in bytes of the rod axis, corresponding to the row or column size for a 2D dataset. The *query rod size* is defined as the length in bytes of the portion that intersects with a specific subset query within a space rod. For example, each 1×4 shaded row corresponds to one query rod in figure 5.5a.

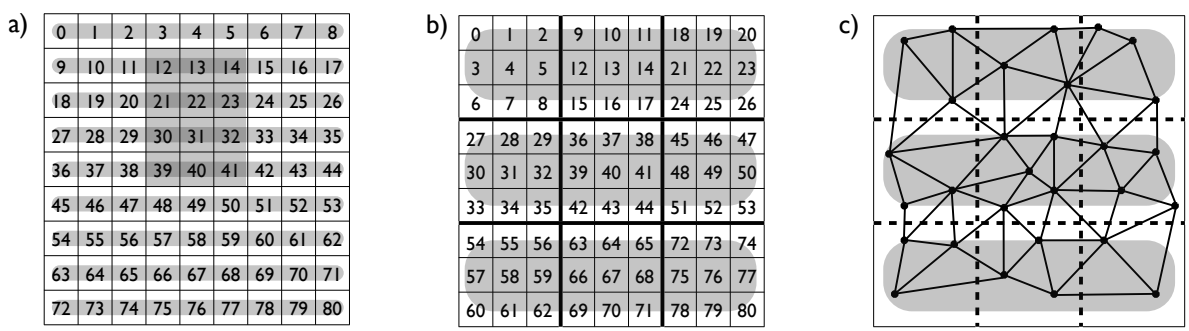


Figure 5.5: Rod storage model can be extended for different types of data organization, including chunking shown in figure (b) and unstructured point data in figure (c).

The rod storage model can also be applied to chunked files [57]. Here, each element of a rod is actually an entire chunk of data corresponding to a block of data values that are nearby in the index space. Because these chunks are stored contiguously in the file, we can read several chunks with one transaction, making the rod storage model useful for predicting performance with chunked files. Figure 5.5b shows a file that has been reorganized into 3×3 chunks, where each row of 3 chunks is stored contiguously, fitting the rod storage model. Here, the same query shown in figure 5.5a would require only 2 read operations, although some extra data would be read.

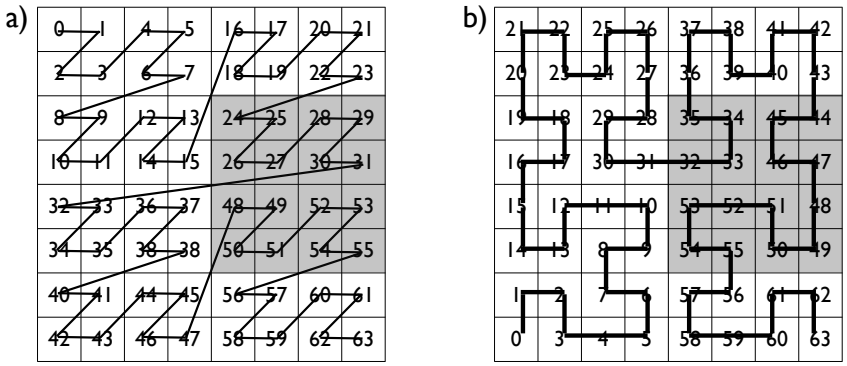


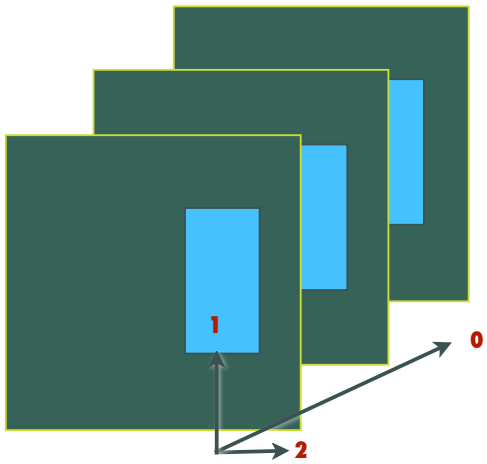
Figure 5.6: Two types of widely used Space-filling curves. Figure (a) shows the Morton Curve and (b) shows the Hilbert Curve. Space-filling curves are widely used to maintain data locality for data access. These two Space-filling curves can be adapted for 3D datasets.

There is no reason why we can't use essentially this same trick with unstructured data, dividing the dataset vertex and cell information into pieces constituting a grid. If grid elements are stored contiguously on disk, even notoriously difficult unstructured datasets can be represented with the rod storage model. Figure 5.5(c) shows a simple unstructured grid of triangular cells in which the geometry has been partitioned into a 3×3 grid. If the data is reorganized so that the cell and vertex data in each row of grid elements can be read with a single transaction, the rod storage model can be used to describe the data. As with the other examples in figure 5.5, this allows us to predict the number of read operations necessary to retrieve the data required for a subblock query.

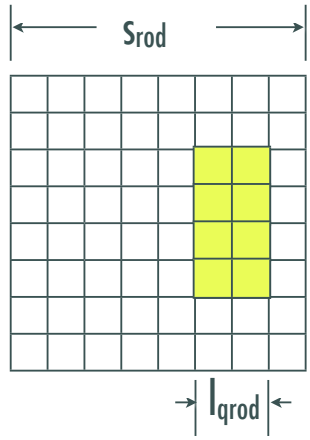
The elements comprising a rod are often just a few values, but could also be whole chunks of rectilinear or unstructured data, allowing the rod storage model to apply to a variety of spatial data formats. Not all data formats can be shoehorned into the rod storage model, however. Space filling curves like the *Morton Curve* [58] and the *Hilbert Curve* [59] have desirable properties for storing spatial data. Developing new storage models to handle these and other file formats is an important avenue for future work.

In particular, elements that are nearby in the data space tend to be nearby in the one dimensional file space. However, it should be clear from figure 5.6 that for these formats, new storage models are needed in order to properly indicate the number of reads, their beginning offsets, and their length. Although we focus on the rod storage model for our testing, the partial replica selec-

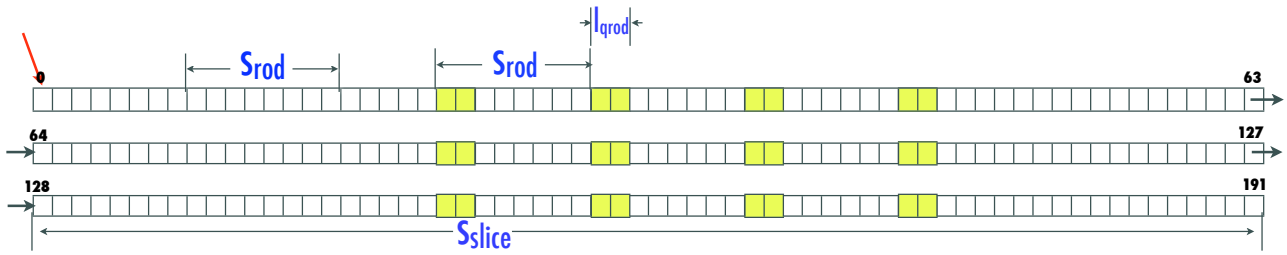
tion algorithm described in section 5.2 can be applied to any storage model.



(a) A 3D dataset with three slices and storage ordering $\{0,1,2\}$. The blue region is a subset query, may or may not penetrate through all slices.



(b) One of slices for dataset in figure (a) and its 2D intersection with the subset query.



(c) 1D file storage for dataset in figure (a) on hard disk. The red arrow is the file pointer. We assume each cell takes one unit of storage. Offset 0 through 63 corresponds to the first slice in the 3D data, offset 64 through 127 for the second slice.

Figure 5.7: An example of access pattern for a subset query.

5.1.2 Access Patterns

An *access pattern* conceptually represents the series of read transactions made to the one-dimensional file underlying the spatial dataset. In the most general case, an access pattern might consist of a set $A = \{(s_0,l_0),(s_1,l_1),...(s_n,l_n)\}$ of integer pairs (s,l) each indicating the *stride* and length of a read transaction. Stride is defined as the difference between the starting offset of the current transaction and the previous one. However, the access patterns generated by the rod storage model can be represented more efficiently, similar to the *regular sections* used to describe access

patterns of nested loops [60]. Looking back at figure 5.2, it is apparent that all reads in a subset query are the same length l_{qrod}, the length of a query rod. Also, an n-dimensional subset query will only use $n-1$ different stride values between reads. In the figure, the rods of each query are grouped into slices, illustrated with alternating colors within the query block. Consecutive rods within a slice are relatively close together, and use a small stride value s_{rod}, which is also the length of a space rod. Moving from the bottom of one slice to the top of the next requires a much larger hop in the underlying file, and requires a second (larger) s_{slice} value. Often,s_{rod} is within range of the filesystem prefetching mechanism, making rod storage access patterns not random, not sequential, but a challenging intermediate case between the two extremes. In any case, we can represent a three dimensional rod storage access pattern with $s_{slice}, s_{rod}, l_{qrod}$, along with l_{slice} and n_{slice}, the length of each slice expressed in rods, and the total number of slices.

In figure 5.7, we describe an example of the access pattern for a 3D dataset with a subset query. Start at offset zero, data server will navigate through the 1D file on the disk in order to answer the subset query. It first locates the first query rod, shown as the first yellow box in figure 5.7(c) on the first slice, followed by a disk read of l_{qrod} bytes. Next it strides length of S_{rod} bytes to locate the second query rod, then reads another length of l_{qrod} bytes. It repeats the same pattern until all query rods are retrieved on the first slice. In order to locate the first query rod on the second slice, we need a larger stride of S_{slice} bytes. Then all the operations on the first slice are repeated for the second and third slice.

The five parameters to describe access patterns for a subset query in figure 5.7 are shown as follows. The space rod size srod S_{rod} is 8 units. The slice size S_{slice} is 64 units. The query rod l_{qrod} is 2 units. Number of query rods on each slice l_{slice} equals to 4 units and number of slices covered by the query n_{slice} is 3 units.

5.1.3 Device Models

While a storage model can indicate how many device transactions a subblock query will require, a *device model* should indicate the cost of those individual transactions. Detailed models

of storage device performance have been an active focus of research for years, and are often used to assist in evaluation of new storage strategies [61, 62, 63, 64]. A complete model should take into account factors like the varying number of sectors in the different zones of a modern drive, how many tracks separate two subsequent accesses, and even the rotational position of the head over the data when a transaction begins.

Such detail is not required for the purposes of replica selection. The model only needs to allow us to choose between the candidates, so as long as it correctly predicts the *relative* performance of different choices, it need not accurately predict absolute performance. For example, a model that overestimated access time by a constant value of one second for all accesses would still allow us to make the right choices between replicas. Furthermore, we do not keep track of cache content because we always proceed in forward direction when reading. In other words, we never revisit pages.

However, a model that is too simplistic is not suitable for predicting the cost of accessing a spatial subset of a larger dataset or replica. Subset operations on spatial datasets generally result in a large number of separate read transactions, so any error in modeling the cost of a read transaction will add up quickly. Our own research [30] has shown the importance of the *filesystem cache* and similar buffering mechanisms along the data path from disk to application. Filesystem prefetching is of particular interest, since a read transaction that accesses prefetched data would be two to three times faster than a similar read directly from disk.

Because filesystem prefetching is such a crucial performance factor, we chose to build this behavior directly into the device model, even though it is part of the operating system. The device model is split into two components, the *raw device model* and the *cache model*. The cache model takes an access pattern as input and returns an estimate of the time required to perform the I/O. It uses the raw device model to estimate access time to the underlying disk device, as shown in a schematic diagram in figure 5.8. We discuss these two components below.

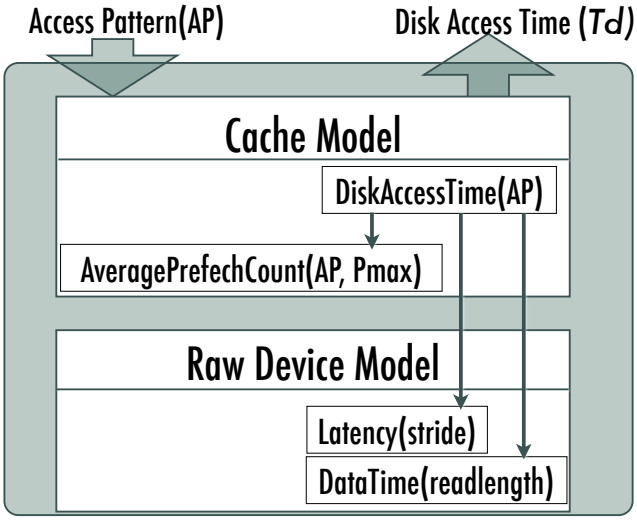


Figure 5.8: A diagram of device model composed of a cache model and a raw device model.

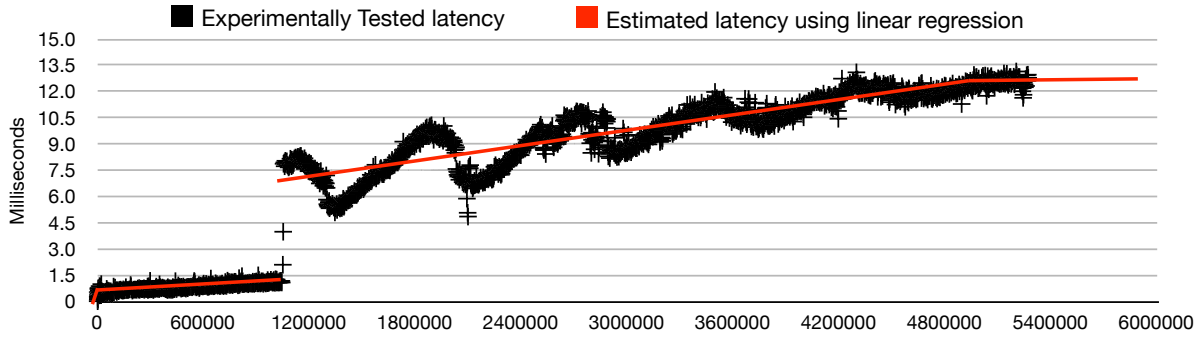


Figure 5.9: The relation between stride size and disk latency. We observe that the disk latency has a sudden jump at 1M bytes of stride , but before that the disk latency is quite linear. If the stride size is greater than roughly 5.0M bytes, the disk latency becomes a constant 12.5 milliseconds.

5.1.3.1 Raw Device Model

Our raw device model implements two queries. $DataTime(readlength)$ is given the length of a read transaction, and returns the time cost of retrieving that number of bytes via available device bandwidth, which is assumed to be constant. Bandwidth values can be derived experimentally or taken from vendor literature. $Latency(stride)$ is given a distance expressed as a change in file offset, and returns a time cost associated with making the device move that distance before reading the next piece of data. This value is determined by evaluating a function similar to the one shown in figure 5.9. The data points in this figure were determined experimentally by stepping through a file with strides ranging from 1 byte to over 5MB. The filesystem cache was disabled. We approximate the resulting function using a piecewise linear function drawn as a line in figure 5.9. The most striking feature of these functions is the sharp jump at 1MB, which corresponds to the size of the *disk cache* segment for the test application. The disk cache is a hardware buffer on the disk controller itself, and should not be confused with the filesystem cache.

5.1.3.2 Cache Model

Conceptually, a cache model takes an access pattern as input, and returns an execution time. If the access pattern is a rod storage access pattern, as described in section 5.1.2, we can take advantage of the compactness of that representation, rather than iterating through a set of separately listed read transactions.

At this level, each slice of the original query corresponds to a stride of length $stride_{slice}$ followed by a sequence of $stride_{rod}, read$ pairs making up the slice itself. This sequence has length $rodsPerSlice$, and may or may not trigger the prefetching mechanism of the filesystem cache.

Filesystem prefetching behavior is determined by the system's settings and whether an application has been accessing the file *sequentially*. If we are currently reading block x and the last read was from either block x or block $x-1$, the filesystem will treat the current access as sequential [65]. If the current access is sequential, the filesystem will double the amount of data prefetched up to some maximum number of (typically) 512-byte sectors. If the current access is

not sequential, the filesystem will cut the prefetch size in half down to a minimum prefetch size. For example, the default settings for the linux *ext3* filesystem specify a minimum prefetch length of $P_{min} = 8$ sectors, and a maximum of $P_{max} = 256$ sectors.

To help model prefetching behavior, we compute an *average prefetch count (APC)*, corresponding to the average number of reads that are prefetched by the filesystem within the slice:

$$APC = \begin{cases} 1 & \text{if } (s_{rod} - l_{qrod}) \geq 2b, \\ P_{max}d/s_{rod} & \text{if } (s_{rod} - l_{qrod}) \leq b \\ simulate() & \text{if } b < (s_{rod} - l_{qrod}) < 2b \end{cases} \tag{5.1}$$

where b is the filesystem block size, d is the size of a disk sector, and the quantity $(s_{rod} - l_{qrod})$ corresponds to the *gap* between the end of one read and the beginning of the next read. In the first case in equation 5.1, this gap is at least twice the block size, so successive reads will never be sequential. The *APC* is 1 because the filesystem is not prefetching. In the second case, the gap is less than one block, so the access pattern is always sequential. Here, the *APC* is simply the maximum number of bytes prefetched divided by the space rod size (which also happens to be the stride between query rods).

The last case in equation 5.1 occurs when the access pattern is sometimes, but not always, sequential within a slice. To generate an *APC* value for this case, we execute a small piece of code which simulates the filesystem's lengthening and shortening of the prefetch length for a representative number of steps of the access pattern.

$$T_d(s) = n_{slice} \times \left[DiskLatency(s_{slice}) + \frac{l_{slice} - 1}{APC} \times (DiskLatency(APC \times s_{rod}) + C_o) \right] + DataTime(size(s)) \tag{5.2}$$

Equation 5.2 computes an estimate of the total disk costs associated with reading the subset volume. The terms inside the square brackets calculate latency costs for one slice using values returned from the raw device model. Recalling that l_{slice} is the number of rods per slice, we can

compute the number of reads needed for one slice as simply l_{slice}/APC, but the first read of the slice is preceded by a large stride s_{slice}. We add $DiskLatency(s_{slice})$ and subtract 1 from l_{slice} in the second term to account for this. We use $APC \times s_{rod}$ to calculate the stride used for the other reads in the slice, accounting for prefetching while reading the slice. A factor C_o is added to account for cache overhead where C_o is defined as:

$$C_o = \begin{cases} 0.1 & \text{if } APC < 2, \\ k & otherwise \end{cases} \tag{5.3}$$

where the value of k is chosen to provide the best fit between our model and experimental data, we set $k = 1.6$ in our experiments. Note that we examine the value of APC to determine if the filesystem prefetching is active. We then multiply this total slice latency by the number of slices, and add the data transfer costs for the entire subset, as provided by the raw device model.

5.1.4 Modeling Replica Transfer Time

While the storage and device models will help estimate the cost of storage access, we must also model network costs and the behavior of servers and clients in a distributed environment. A complete model of replica transfer time would take into account the varying number of servers sending data to a client, changes in network bandwidth and latency, and the use of multiple disk devices on each server. For our current work, we have made some simplifying assumptions. Each server is assumed to store all of its data files on a single storage device. We assume that both network bandwidth and latency are constant between any pair of nodes, and that they have been recently measured with tools like *iperf* and *ping*. We assume that all servers that are transferring data to a client have an equal share of available client bandwidth. Lastly, we assume that if we add up those shares, the sum will not be greater than the bandwidth that would be consumed by a single server. This last assumption doesn't always hold for TCP traffic, but we are using UDT [34] for data transfer, which efficiently uses available bandwidth even with a small number of servers. The modeled network cost $T_n(s \subseteq r)$ to transfer the subset s of replica r residing on *Server* to a

Client is shown in equation 5.4.

$$T_n(s) = \begin{array}{l} size(s)/NetworkBandwidth(Client, Server) \\ +NetworkLatency(Client, Server) \end{array} \tag{5.4}$$

The total cost $T(s)$ to access subset s consists of the disk access cost and the network cost:

$$T(s) = T_d(s) + T_n(s); \tag{5.5}$$

Taken together, these various assumptions support an additive model of transfer time. Assume that $T(s \subseteq r)$ is the time taken to transfer a subset s of a partial replica r from its host to the client that issued the request, estimated using equation 5.5. Computing the transfer time for the set of all such subsets $H_{host} = \{s_0 \subseteq r_0, s_1 \subseteq r_1 ... s_n \subseteq r_n\}$ involving a given host is simply:

$$T(H_{host}) = \sum_{i=0}^{n} T(s_i); \tag{5.6}$$

This model assumes there is no opportunity to overlap transfer costs of the various subset queries on a single host. Even though we use multiple server threads to simultaneously satisfy multiple queries, the execution time will be the same as if we had serviced each query one after the other.

There is nothing preventing overlap of transfer costs on *different* hosts, however. To compute the overall transfer time $T(Q)$ of a spatial query Q, we must first identify the set of hosts $H = \{H_0, H_1 ... H_m\}$ containing the replicas that will be used to satisfy the query. $T(Q)$ can then be written as:

$$T(Q) = max(T(H_0), T(H_1), ... T(H_m))) \tag{5.7}$$

That is, we assume that the various hosts involved in a query can work independently, and that the total query execution time will be determined by the host that takes the most time to complete its work. We call this host the *bottleneck* host. Since $T(Q) = T(bottleneck)$, finding the best solution to the replica selection problem will require minimizing $T(bottleneck)$ by distributing load among

available hosts.

$$goodness(s_i \subseteq R_i) = \frac{Size(s_i) * NetworkBandwidth(Client_i, Host_i) * DiskBandwidth(Host_i)}{NumReads(R_i) * NetworkLatency(Client_i, Host_i) * DiskAverageSeekTime(Host_i)} \tag{5.8}$$

```
INPUT: Query Q; a set of partial replicas S that intersected with Q.
OUTPUT:  S', a subset of S, so that S' can answer the query Q and the
         transfer time of S' is minimized.

S' = {} //empty set
St = Sort(S) //sort S according to goodness of replica in descending order
initialize(C)  //use C to store a collection of selected replicas
while St not empty
    Replica r = getFirstElement(St)
    St = St - r
//Let u be the replica in C that matches r.
//Note that C contains no more than one match for r.
    u = C.match(r)
    if u does not exist
        C.insert(r)
        S' = S' + r
    else
        S* = S' - u // temporary set S*
        S* = S* + r
        if  ( T(S') > T(S*) ) or ( T(S') == T(S*) and T(r) < T(u) )
            C.delete(u)
            C.insert(r)
            S' = S*
        end if
    end else
end while
```

Figure 5.10: Partial spatial replica selection algorithm. For replicas that partially overlap Q, we only count the overlapped region for transfer cost.

5.2 Partial Spatial Replica Selection

Our partial replica selection algorithm chooses among all replicas intersected with a spatial query and chooses a subset of those replicas, so that the total transfer time for that subset is minimized. It must take into account storage organization, network performance and load balancing, so that we may spread the load among many grid nodes.

5.2.1 Partial Replica Selection Algorithm

The algorithm is described in figure 5.10. Currently, we make the simplifying assumption that if two replicas overlap, they have the same MBR. Such replicas will represent the data using different storage orderings, and will likely be on different hosts, so choosing between them will have a significant effect on performance.

Given a subset $s_i \subseteq R_i$ of replica R_i stored on $Host_i$, the *goodness* value of s_i can be calculated using equation 5.8. This value can be seen as a ratio of beneficial and detrimental factors for s_i, somewhat resembling the goodness value employed by Weng, et al. [66].

Replica subsets are first sorted according to their goodness value in descending order, and we initialize an empty collection C. The main loop of the algorithm (lines 4-20) iterates through the list of sorted subsets, and checks whether each replica subset r has an MBR that matches a replica u already present in C. If there is no match for r (i.e. u does not exist), r is simply added to C. It is also added to a set S' that contains all replica subsets present in C. If r does match with some u already present in C, we must decide whether to add r to C, or simply keep u, depending on which choice provides faster performance. We do so by first constructing a set S^* that contains r and all the replica subsets that are currently in C except u. We can now compare the performance S^* with S', as seen in lines 12 through 14. If the performance of S^* is better than that of S' (i.e. $T(S') > T(S^*)$), or if performance of S^* is equal to that of S' and the transfer time cost of r is less than that of u, we use replica subset r instead of u in collection C (and S'). Otherwise, we leave the collection C unchanged. This is shown in lines 14 through 18. After the whole list of sorted replicas are processed, the set of replicas in collection C or S' is the replica selection *solution*.

Note that because we do not yet handle replicas with arbitrary overlap, if u intersects at all with a replica r, it will have exactly the same shape as r. For this reason, a spatial data structure is not strictly necessary for C, and we can use a hash table to detect exact MBR matches. However, in future work we will handle the more complicated scenario where u and r only partially overlap, which requires an effective spatial indexing structure. With this in mind our current implementation uses an R-tree for the collection C.

Given n replicas intersected with query Q, if we use an R-tree for C, the R-tree query and insertion operation take $log(n)$ time on average, making the total complexity of the algorithm $O(n\ log(n))$. Using a hash table for C reduces complexity to $O(n)$.

In our discussion of equation 5.7, we introduced the concept of a bottleneck host. As we choose replicas and insert them into the collection C, step 14 ensures that the bottleneck host changes as the algorithm progresses and that the load is spread among many grid nodes. For example, if we constantly try to insert replicas from a single grid node H_k into C, then H_k will eventually become the bottleneck host. When this happens, step 14 will always choose a replica from an alternate host H_j if one is available. After choosing this alternate, $T(S')$ can not be larger than it would have been if H_k had been chosen again, since $T(H_j)$ was originally less than $T(H_k)$.

Despite the small chance of model failure shown in section 5.3.2, the excellent selection results that our replica selection algorithm finds can be attributed to the algorithm combining a local heuristic search and global load balancing. As described above, the selection algorithm examines the replica with a maximal goodness value for a local region. Meanwhile the algorithm considers how the time cost of an individual replica affects the total time cost for the current set of replicas that had already been selected. The latter is a global load balancing operation.

Table 5.1: Grid Node Characteristics

OS	Processor	Cores	Memory	Hard Disk	File System	UDT	Java version	Globus Toolkit
Linux 2.6.18	Intel Xeon 2.40GHz	16	24G	1TB	ext3	v4.10	SE 1.6.0_23	v5.0.3

Table 5.2: Network Parameters Between A Client at Mississippi And Various Data Servers

	Univ. Mississippi	Univ. Florida	Univ. Chicago	Univ. Notre Dame
Bandwidth	746Mbit/s	735Mbits/s	724Mbits/s	718Mbits/s
Latency	0.28ms	52ms	41ms	95ms

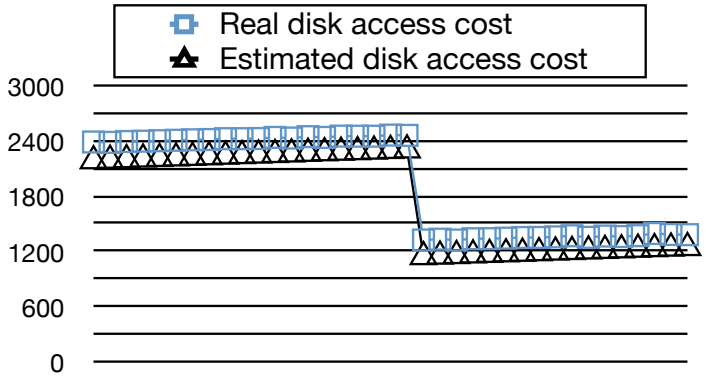
5.2.2 Implementation

For the collection C mentioned in figure 5.10, we used an existing R*-tree [42] implementation [67] and added a few features for our own purposes. These features include adapting the existing implementation to accept our Replica object, and adding a method for removing a replica from the tree. Since this implementation is an out-of-core R*-tree implementation with a user-specified in-core cache size, it is able to handle a large number of replicas. We refer to this implementation as *R-tree Based Replica Selection* in section 4.4.

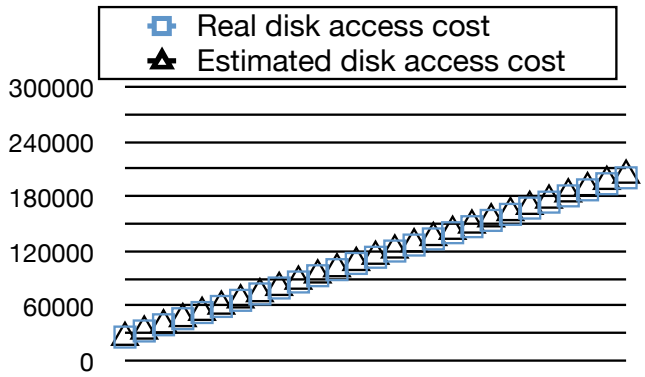
We introduced a *Repository* class to keep track of replicas that have already been added into the R-tree. A Repository object provides several services. First, when we add or remove a replica, the bottleneck and total transfer time will be updated. Second, it can map a host name to a list of replicas stored on that host, and map a *hostname* to $T(hostname)$. Lastly, a Repository can return the current bottleneck and total transfer time for the replicas currently in the repository. We use a hash table for the Repository class, so it does not disturb the overall complexity of the algorithm.

5.3 Experiments

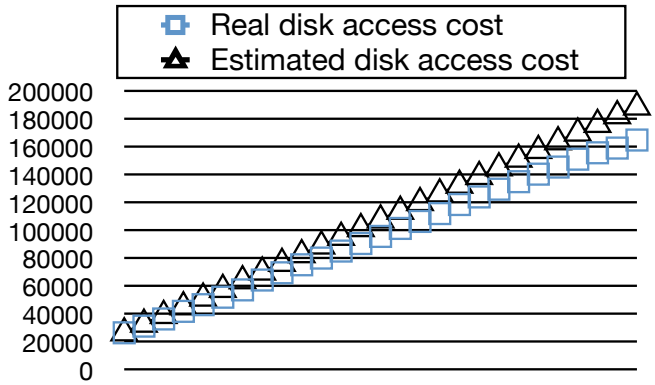
The *Distributed Research Testbed (DiRT)* is a multi-site instrument for developing and evaluating the performance of distributed software. We constructed a grid on *DiRT* using a version of Globus Toolkit 5.0.3 modified to include our spatial replica location service(SRLS). The characteristics of each grid node are described in table 5.1. Because we use a client machine at the University of Mississippi, we show the UDP bandwidth and network latency between this client and different data servers located at different universities in table 5.2.



(a) Query MBRs start at {0,0,0,50,50,50}, each query increments by 50 the largest coordinate of the previous query MBR on axis 2, in order to change the query access rod size.

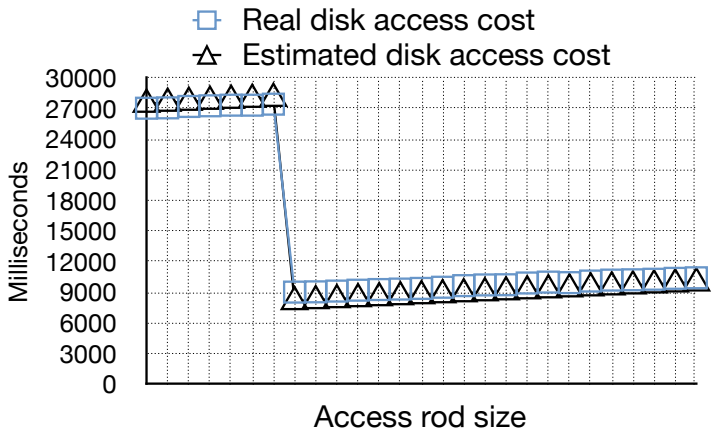


(b) Query MBRs start at {500,500,500,700,700,700}, each query increments by 50 the largest coordinate of the previous query MBR on axis 0, in order to change the number of slices a query extends.

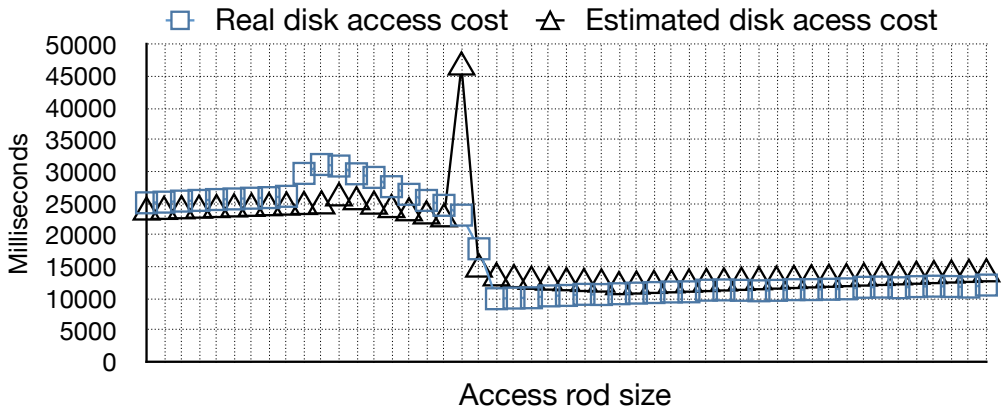


(c) Query MBRs start at {500,500,500,700,700,700}, each query increments by 50 the largest coordinate of the previous query MBR on axis 1, in order to change the number of query access rods on each slice.

Figure 5.11: Device model verification using three sets of expanding queries. The time cost is measured in milliseconds.

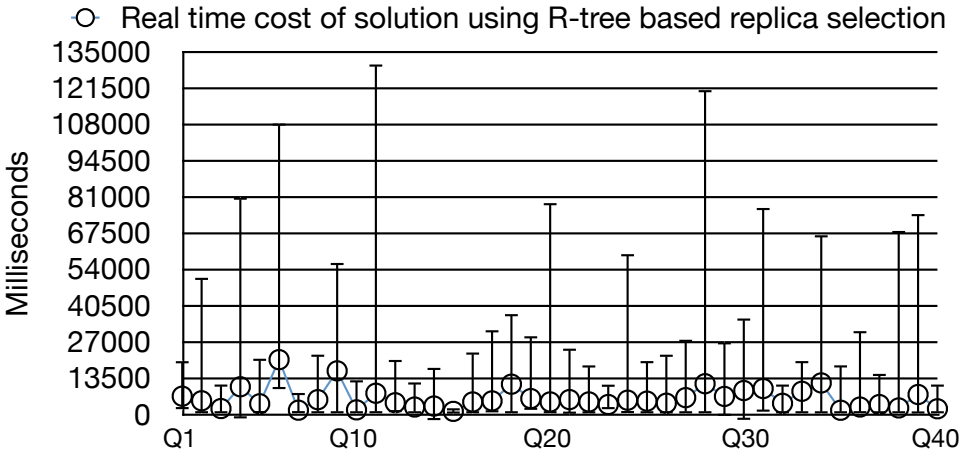


(a) Query MBRs start at {500,500,500,700,700,700}. Each query increments by 50 the largest coordinate of the previous query MBR on axis 2, in order to change the query access rod size. The space rod size is a multiple of page size of 4KB.

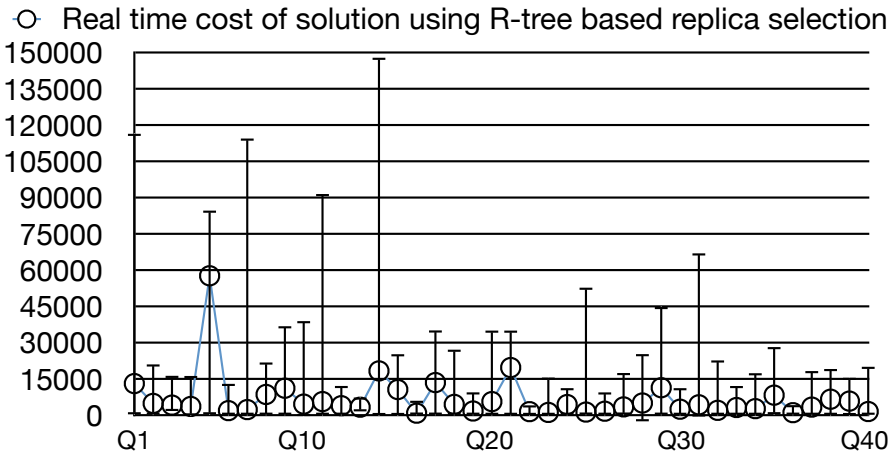


(b) Query MBRs start at {900 750 900 1100 950 950}. Each query increments by 50 the largest coordinate of the previous query MBR on axis 2, in order to change the query access rod size. The space rod size is NOT a multiple of page size.

Figure 5.12: Device model verification using two sets of expanding queries. The time cost is measured in milliseconds.

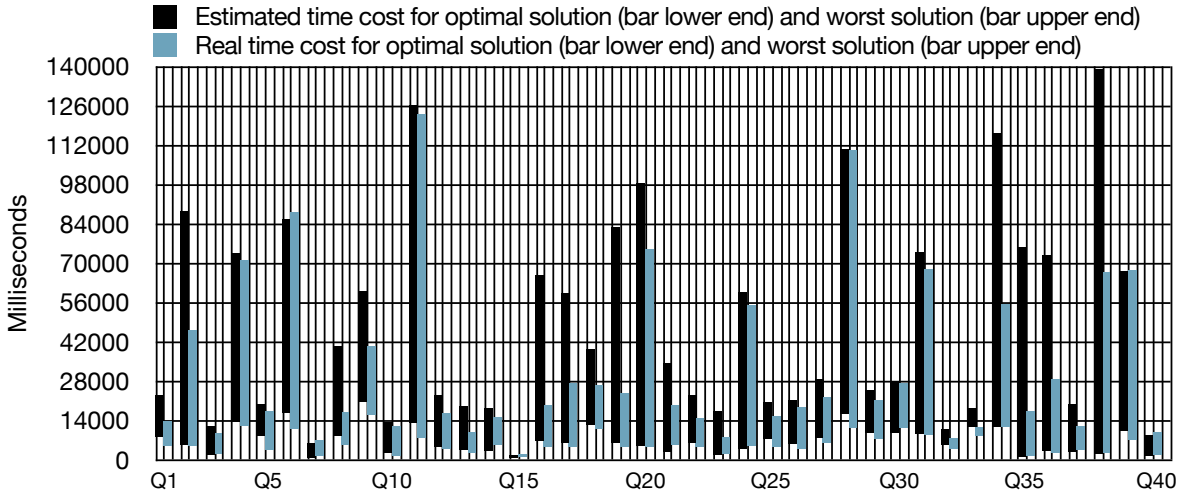


(a) Real time cost of R-tree based, optimal and worst solution using 4 nodes.

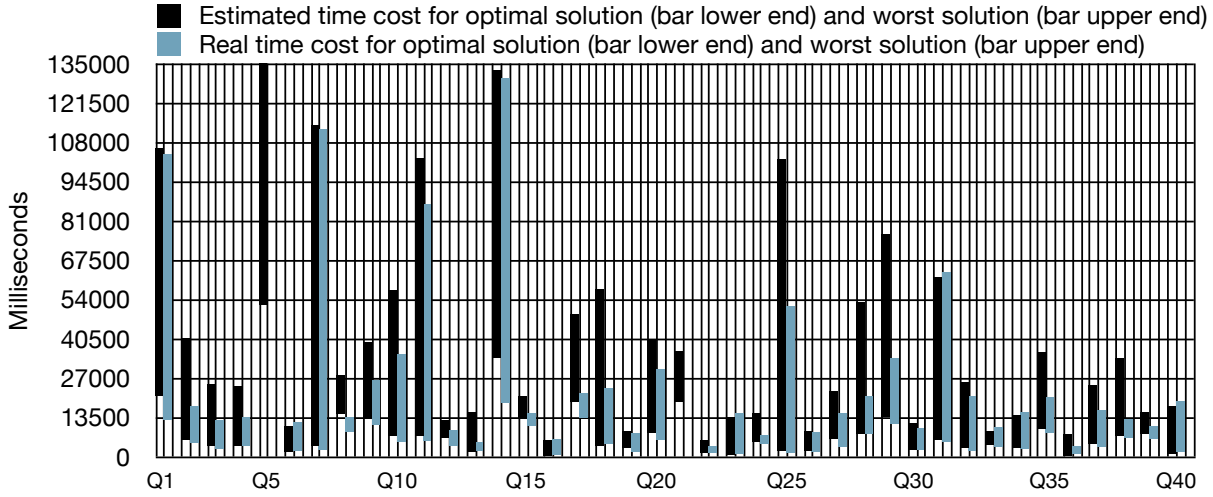


(b) Real time cost of R-tree based, optimal and worst solution using 8 nodes.

Figure 5.13: Verifying replica selection algorithm using 4 and 8 grid nodes. The high and low end of the vertical line segments show the real time cost for the worst and optimal solution found by exhaustive search. The circle on the line segment shows the real time cost of the solution found using R-tree based replica selection. The horizontal axes in (a) and (b) denote two different sets of forty random spatial queries. Times are shown in milliseconds.



(a) Real time cost compared with estimated time cost using 4 nodes.



(b) Real time cost compared with estimated time cost using 8 nodes.

Figure 5.14: Verifying replica selection algorithm using 4 and 8 grid nodes. We compared the real time cost with the estimated time cost for the worst and optimal solutions. Note that the set of queries and datasets used in 4-node and 8-node tests are completely independent and random.

In order to verify the device model and our replica selection algorithm, we designed experiments with two different goals. First, we have to verify that the device model and storage model can estimate the disk access cost accurately. Second, with the help of the storage and device models, we have to verify that our algorithm can choose the right set of replicas. We compared solutions with the optimal solution found by exhaustive search, and also noted execution time for the selection process itself. Exhaustive search is used only to establish the real-world optimal and worst selection solution, and not for comparing time performance of the selection algorithm. In all tests, we used the Granite scientific database as data servers, which in turn uses the *UDT4* protocol for data transfer.

5.3.1 Verification of Device Model and Storage Model

We tested the device model in two different scenarios. First, we used a spatial dataset with a space rod size that is a multiple of the filesystem block size, and $MBR = \{0,0,0,2047,2047,2047\}$. Figure 5.11(a) and figure 5.12(a) show the results for several tests in which we extended the shape of a spatial query along the rod axis. In figure 5.11(b), we start the query at a new location and extend the shape of a spatial query to cover more slices in space domain. In figure 5.11(c), we expand the query rectangle so that it may enclose more query access rods on each slice that intersects with the query. The results show a close match, even with the sudden transition when prefetching activates, which we call the "jump edge".

In a second test, we designed a dataset whose space rod size is not a multiple of page size, with $MBR = \{900,750,900,1500,2750,3399\}$. In figure 5.12(b) we see that the real device transitions more gradually than the model as prefetching becomes active, but the match is still reasonable with one exception. In both scenarios, we also tested various queries by incrementing the number slices or the number of query rods of one slice, with varying query locations in the data domain. All results show a close match except for a very few errors under the second scenario.

In figure 5.12(b) where the space rod size is not a multiple of page size, our device model has a roughly 100% error for a query that falls exactly upon the jump edge. This error is due to the

fact that we only have two values for the cache overhead C_0. In equation 5.3, we use C_0= 0.1 when $APC < 2$, corresponding to when prefetching is not active. Otherwise C_0=1.6, corresponding to the active prefetching case. That is, we are using a simple step function to relate C_0 to APC. However, the real disk behavior in figure 5.12(b) shows more gradual behavior in which prefetching becomes increasingly active over a range of APC values. For example, the query with the big error in figure 5.12(b) has $APC = 2.03$, but the real filesystem prefetching has not been activated yet. In this case, our device model erroneously treats it as active prefetching when using APC threshold 2.0, which in turn uses the cache overhead $C_0 = 1.6$ and produces a large error for this query. Despite this problem, we find that using $APC = 2.0$ as the threshold in our step function yields better overall results in our experiments.

5.3.2 Verification of Replica Selection

We randomly synthesized all the partial replicas within a given 3D data domain with $MBR = \{0,0,0,8192,8192,4096\}$ for a four-node test, and $\{0,0,0,8192,8192,8192\}$ for an eight-node test. The total number of partial replicas in our grid is 3072, with size of replicas ranging from tens of megabytes to several Gigabytes. The four node test involves a total data volume of 3.07TB, while the eight node test uses 6.14TB.

We generated metadata of all partial replicas in a centralized fashion, partitioning the entire dataset MBR into smaller MBRs, and then randomly declustering these smaller MBRs among the grid nodes used in the test. Each smaller MBR results in three partial replicas with different storage orderings. The storage orderings in the test are: {0,2,1}, {1,0,2}, {2,1,0}, as described in section 5.1.1. For any arbitrary point in the domain, there exists three and only three partial replicas that contain this point. These three partial replicas have the same MBR, but have different storage orderings and are probably stored on different grid nodes.

To randomly generate a set of arbitrary MBRs which can fill in the given data domain, we used a recursive top-down partition algorithm, similar to the K-d tree construction algorithm [68]. We have a list L as an input to the algorithm, which initially only contains the MBR of entire data

domain. Then we partition each MBR in L into two MBRs, generating a new list L', with a random cutting-plane perpendicular to one axis i. We recursively call the partition routine with input $L = L'$ and axis $i = (i+1)$ *mod Dim*, until specified number of partitions are completed, where *Dim* is the dimensionality of the data domain. During partition, we set several rules to avoid the small slit MBRs, while ensure MBRs have various shapes.

We compared our selection algorithm with an exhaustive search selection algorithm that can find the real optimal and worst selection solution. We randomly generated the location of test queries in the whole data domain and the dimension along each axis between 10 and 700. We cleaned the filesystem cache between each query on the data servers.

In figure 5.13, we show that our selection algorithm can find a solution that has excellent performance in nearly all tests. Examining the real time cost, we observed that the solution found using our algorithm is as good or better than 98% of solutions found by exhaustive search, and performance is on average always at least 91% and 93.4% of the real time cost of the optimal solution in 4 and 8 node tests respectively.

Our replica selection algorithm scales well as we double the size of dataset and the number of grid nodes from 4 to 8 nodes in real-world tests. In addition to these tests, we also performed a simulation using 64 nodes and metadata for 546,000 replicas to verify system behavior with a large number of nodes and replicas. We obtained similar performance results as in the 8-node tests, largely because the vast majority of replicas are discarded in the replica location stage, leaving only a few thousand to be considered by the replica selection algorithm. The execution time for the R-tree based selection implementation takes roughly three seconds when the number of replicas intersected with a query is six thousand.

Figure 5.14 shows in most cases the estimated time cost matches well with the real time cost for the optimal solutions, the lower end of bars shown in the figure. For the worst solutions, even though they are less exactly matched, their fluctuation trends are very similar. Of course, matching the worst solution is not a goal of replica selection.

We observed an average of 3% of model failure during our tests as shown by queries Q5

and Q21 in figure 5.14(b). First, we verified that the model failure is not because of the increased number of grid nodes. On the contrary, we observed that these failures are more likely when a query is serviced by only one partial replica. The failure is actually caused by our device model error shown in figure 5.12(b), in which the device model overestimates the disk access cost by roughly 100% for the query falling upon the jump edge. In future work, we will address this very rare case by using linear interpolation instead of a step function to relate cache overhead to *APC*.

5.4 Chapter Summery

We describe a fast replica selection algorithm that provides load balancing and takes file storage organization, filesystem prefetching and hardware performance into account using separate models. By comparing with the optimal solution found by exhaustive search, we observed that the solution found using our algorithm is as good or better than 98% of solutions found by exhaustive search selection, and performance is on average alway at least 91% and 93.4% of the optimal solution in 4-node tests and 8-node tests respectively. The experiments indicate that it is feasible to solve the partial replica selection problem in $O(n\ log(n))$ time, while providing results with quality that is more than adequate.

There are several avenues for future research. Adding more storage models should allow us to accommodate a wider variety of datasets. We may replace our static network model with a grid *Monitoring and Discovery System (MDS)* [69] to address fluctuating network performance. Also, we can improve our device model to eliminate the error when a query falls upon the jump edge shown in figure 5.12(b). Most importantly, we must also address the more general replica selection scenario in which replicas may overlap in an arbitrary way, adding considerable complexity and requiring the special properties of an R-tree or similar data structure.

CHAPTER 6

RELATED WORK

Replica selection in grid computing has been addressed by many researchers. Chervenak et al. described the *Giggle* framework, including RLS requirements, implementation, and performance results [19]. In another paper, Chervenak et al. present the implementation and the evaluation of the RLS [20] in Globus Toolkit [70, 71, 72, 73, 74]. Cai et al. proposed a *Peer-to-Peer Replica Location Service(P-RLS)* with the properties of self-organization, fault-tolerance and improved scalability [21]. More recently, Chervenak et al. systematically described Globus RLS framework design, implementation, performance and scalability evaluation, and its production deployments [4].

Rahman et al. [26] proposed two different replica selection techniques for a grid environment. First, a k-Nearest Neighbor rule is used to select the best replica for a logical file using information in a file transfer log. The author also proposes a technique to predict the file transfer time using a neural network, considering three factors: the historical grid transfer time, current network bandwidth, and file size.

Krishnamurthy et al. use estimates a replica's response time distribution based on performance measurements regularly broadcast by the replica. An online model uses these measurements to predict the probability with which a replica can prevent a timing failure for a client [75]. Lu et al. present a method for constructing a hierarchy of partial replicas from a collection where each replica is a subset of all larger replicas, and extend the inference network model to rank and select partial replicas [76]. Ferdean et al. present a replica selection strategy that adapts its criteria dynamically so as to satisfy application providers' and clients' requirements, aggregated into an application binding contract and a client binding contract respectively [77].

Vazhkudai et al. [1] present a design and implementation of a high-level replica selection service, which uses information about replica location and user preferences, and suggest that Condor's *ClassAds* mechanism is effective for representing and matching storage resource capabilities and policies. The *PU-DG Optibox* package is proposed by Li et al. [27] to facilitate parallel download from a ranked list of grid nodes with higher download performance, using three factors: network bandwidth, transmission distance and download history. Zhao et al. presented *GRESS*, a grid replica selection service based on the Open Grid Services Architecture, which unifies many replica selection methods into a generic platform [2]. In GRESS, a light-weight algorithm is used as the default replica selection module, which is built on *Instance-based Learning (IBL)* technology, and uses transfer log files.

The *Gfarm* grid file system described by Tatebe et al. [3, 78], supports complete file replication at multiple locations to load-balance heavy access to the same file. To choose a replica among replicas on several I/O servers, it uses CPU load and available capacity on I/O servers and a Round-trip Time (RTT) between the I/O server and the client.

In the GIS context, Wu et al. described a framework for a spatial data catalog implemented on top of a grid and P2P system [23]. In the *JXTA*-based three-tier framework, the logical layer is partitioned into physical fragments, and the fragment layer is accessed through the node's *Spatial Database*. In another paper, Wu et al. described *Barn*, a spatial data grid prototype based on *WSRF* [79]. Wei and Di et al. implemented a Grid-enabled catalogue service for GIS data, by adapting the *Open Geospatial Consortium (OGC) Catalogue Service for Web Specification* and the *ebXML Registry Information Model (ebRIM)* [24, 25]. The OGC publishes a number of standards for GIS databases and applications, but these aren't well suited for other types of spatial data, especially those that entail three or more dimensions. For this reason, databases such as PostGIS [80] that implement the OGC GIS standard are not easily applied to other types of scientific data or metadata.

To manage spatial metadata for fast retrieval, we chose the R-Tree [41, 42, 81, 82, 83] data structure for our Spatial Replica Location Service (SRLS) from among several other spatial data

structures, including the Quadtree, Octree, Kd-tree and UB-tree. We require a data structure which returns the collection of replica *Minimal Bounding Rectangles(MBRs)* that intersect with a query region. OLAP methods like the UB-tree [84, 85, 86, 87] return a collection of *points* (records) that are contained within a query region, making them inconvenient for our purposes. As described by Samet et al., Quadtrees and Octrees typically divide the spatial replicas into smaller blocks, thus generating more partial spatial replicas during construction of the tree [88, 89, 90, 91]. Kd-trees [68, 92] have a similar problem because they require us to divide the space with a splitting hyperplane. However, the R-tree and related methods (e.g. the X-tree [93]) allow for efficient retrieval of the replica MBRs that intersect with a query region, without splitting replicas into smaller pieces.

Kamel et al. propose an early version of parallel R-tree, initially designed for multiple disk systems to maximize the system's throughput [45]. Three approaches to distribute a R-tree on multiple disks are proposed: the d independent R-tree, super-nodes, and the Multiplexed R-tree. In our approach, the GTR-tree uses a structure similar to the d independent R-tree, with each grid node storing a whole chunk of sub-tree independently, so that the inter-node communication will be minimized during the query.

The Master-Client R-tree, another parallel-R-tree architecture proposed by Schnitzer et al., is an R-tree existing on both master and client nodes [40]. However, there still exists a sequential computation to traverse the R-tree on the master node. That is, the client nodes will not perform a parallel query autonomously until the master completes traversing its R-tree. Our work differs from the Master-Client R-tree in two aspects: first, our GTR-tree has a multiple-master structure, similar to a P2P shared-nothing architecture [47]; second, to maximize the parallelism, after the Client Node conducts the intersection test for each grid node, grid nodes with the root MBR intersected with the query will be invoked to perform the spatial query independently.

In order to facilitate the spatial join on a parallel R-tree architecture, Mutenda et al. propose the Replicated Parallel Packed-R-Tree structure (RPP-R-Tree), replicating the whole R-tree structure across all the slave nodes to reduce the bottleneck caused by the Master R-tree as each client

has to wait for the Master to complete traversing two joining trees [46]. Because we distribute the replica metadata among many grid nodes, our system is more scalable.

We also use the *Morton Space-filling curve*, also known as *z-ordering*, in order to construct a more effective R-tree. The Morton Space-filling curve and *Hilbert curve* are widely used in multidimensional access methods [43, 94]. Faloutsos et al. compared the Hilbert curve and the Morton curve. Their work concluded that the Hilbert curve has better distance-preserving mapping than the Morton curve, but the Hilbert curve is more complex to calculate [52]. Both the Morton and Hilbert curves are found in commercial database systems. *Microsoft SQL* server 2008 used the Hilbert curve to determine the value of cell identifier, thus maintaining good spatial locality [95]. *Oracle* has used the Morton curve in their products for some time [94]. *TransBase*, a relational database system, integrated the *UB-Tree* into its kernel to support spatial indexing. A UB-tree combines a *B-Tree* with the Morton curve, mapping multi-dimensional data to one-dimensional space using the Morton value of the data points [85]. In P2P systems, Ganesan et al. [96] used the Morton curve to reduce multi-dimensional data to one dimension, then partitioned the dataset according to contiguous Morton value range. Because the Morton value can be simply computed by bit-interleaving, in this work we use the Morton curve to re-order the 3D replicas and then construct the *Mortonized Aggregated Query R-tree (MAQR-tree)*, as described in section 4.3.2.

The MAQR-tree representation in a relational database uses the concept of *data-linearization* [97][98]. In order to prefetch pointer-linked data structures, Luk et al. proposed two solutions: data-linearization prefetching and jump pointers. Data-linearizaion prefetching removes pointers by arranging data in a contiguous segment of memory. The *Cache Sensitive B+-Tree (CSB+Tree)* employed the same idea, having only one pointer to a group of contiguous child nodes in memory [99]. In this work, we use integer keys as references to a tree node and store the sibling nodes contiguously in a database table, but allow tree updates as described in section 4.3.2.2. This design is very suitable for prefetching.

I/O Prefetching is an important technique for improving performance by overlapping data access and computation costs. Its effectiveness is greatly enhanced when guided by advance

knowledge of what data is needed, then loading the data before they are accessed. Patterson et al. describe *Informed Prefetching*, in which application-disclosed access patterns(hints) are used to prefetch blocks [100]. Rhodes et al. describe *Iteration Aware Prefetching* in which the application uses iterators to describe the access pattern to underlying middleware, which then performs prefetching on behalf of the application [30][31].

Tree prefetching has been investigated in many works, but mainly focuses on the trees that reside in main memory [101][102][103][104]. Chen et al. evaluated prefetching *B^+-Trees (pB^+-Trees)*, which uses prefetching to effectively create wider tree node that could occupy multiple cache lines. In pB^+-Trees, these cache lines comprising a wide tree node are prefetched in parallel with the first cache line in the tree node, thus hiding the latency of cache misses [101]. Kang et al. described R-Tree prefetching for the *SPR-tree (Selective Prefetching R-tree)* [103], which used an idea similar to the pB^+-Trees. Namely, the SPR-tree prefetches all the cache lines that comprise the wider tree node for an R-tree in main memory. In order to optimize both cache and I/O performance, in another paper Chen et al. proposed the *fractal prefetching B^+-Trees (fpB^+-Trees)*. The fpB^+-Trees embed cache-optimized trees(pB^+-Trees) within disk-optimized trees [102]. Kim et al. described fast architecture sensitive binary tree search on modern CPUs and GPUs, in which they use prefetching to overlap the memory access and computation for different queries [104]. In comparison, our out-of-core R-tree prefetching (RTP) utilizes the explicit knowledge of the access pattern required by the R-tree breadth-first search, and prefetches tree node data using a separate I/O thread.

Out-of-core tree traversal has been explored by many researchers. Lindstrom used out-of-core techniques with an octree for large surface meshes to visualize multi-resolution surfaces [105]. However, prefetching was not used during tree traversal. Sulatycke et al. used a tree traversal thread with several computation/visualization threads in order to hide disk latency costs, but also did not use prefetching during tree traversal [106]. The work described here applies prefetching to the problem of out-of-core R-tree traversal when processing a spatial query.

The replica selection problem has been a popular research topic in cluster computing and

grid computing, although *spatial* and *partial* replica selection have received less attention.

Chang et al. [8] presented an algorithm for fragmented replica selection and retrieval in a data grid. Their fragmented replica selection procedure chooses blocks among many sites so that parallel downloading time can be minimized. The authors assumed that the downloading time is the same for each block, regardless where the block is stored. Their algorithm has a complexity of $O(n^2)$, where n is the number of blocks of data to be accessed. In contrast, the work described here takes into account the network performance between various grid nodes, and our selection algorithm has lower complexity.

Narayanan et al. describe *GridDB-Lite*, a middleware package which provides basic scientific database support for data-driven applications in a grid environment [107]. In another paper, Narayanan et al. [28] proposed a runtime framework to address the partial spatial replication problem, in which a portion of the dataset is extracted, re-organized and re-distributed across the storage system. However, the query algorithm used by Narayanan to find the set of chunks to satisfy the query is $O(m \times n)$, where m is the total number of partial replicas in the system and n is the average number of chunks contained in one replica.

Weng et al. described a partial replica selection algorithm for serving range queries on multidimensional datasets [66, 7]. In this work, datasets are partitioned into partial replicas (either space-partitioned replicas or attribute-partitioned replicas, or both), with each replica further partitioned into many chunks. Each chunk is distributed across many cluster nodes to maximize parallelism. The author established a cost function to compute the cost of a fragment, which considers the number of disk blocks involved, average disk seeking time and average disk read speed. The replica selection algorithm has a complexity of $O(n^2)$, where n is the number of chunks intersected with a range query. Song et al. presented a model to estimate data access cost of different data layout policies and to calculate the overall I/O cost of any given application [108]. None of these works take into account the effect of the underlying file system's caching and prefetching, which can dramatically influence the number of disk reads for a specific access pattern, and thus the total I/O time cost [65].

Our work differs from these others in many important respects. First, we divide datasets into *partial spatial replicas* of arbitrary size rather than fixed size *chunks* as used in Weng's and Narayanan's work. Second, our performance models include network costs that differ between hosts, rather than assuming all replicas are equally expensive to access, as in a cluster environment. More significantly, our work will take into account the filesystem caching and prefetching behaviors when modeling file transfer time. Lastly, our replica selection algorithm uses a heuristic to achieve a complexity of $O(n\ log(n))$, a considerable improvement over previous efforts.

CHAPTER 7

CONTRIBUTIONS

In this chapter, we summarize the contributions that has been made in this work. First, we describe two designs for a *Spatial Replica Location Service (SRLS)*, the GTR-tree and MAQR-tree implementation, which is used to register and discover partial replicas in a grid. The SRLS quickly returns a list of partial replicas that intersect a spatial range query. Integrating a relational database, a spatial data structure and grid computing software, we build a scalable solution that works well even for several million replicas. The fast SRLS is implemented using a modified *Globus Replica Location Service*, but can be easily adapted to other grid computing systems.

Second, we have also added a collection of optimizations that together improve performance by a substantial margin. Namely, we have improved upon the *GTR-tree* by changing the table design in the backend database, and by aggregating several queries into one larger query, which reduces overhead. We also use the *Morton Space-filling Curve* during R-tree construction, which improves spatial locality. Lastly, we describe *R-tree Prefetching(RTP)*, which effectively utilizes the modern multi-processor architecture. *RTP* is designed to overlap the CPU execution time with the I/O time, therefore improving performance.

Third, we propose a cost model that can accurately predict for which access patterns the operating system will trigger its prefetching mechanism. Operating system prefetching causes the access time cost to vary by a factor of 2 to 3. We use a *storage model*, from which we can derive the disk access pattern for a query. We then estimate the time cost for the query, considering the operating system's caching and prefetching and hard disk behavior.

Lastly, we describe a greedy heuristic algorithm for the partial replica selection problem, which has been shown to be NP-complete [8]. Our partial replica selection algorithm takes into

account storage organization, operating system prefetching, hard disk parameters, network performance and load balancing. The selection results are expected to be more accurate with these separate models.

CHAPTER 8

CONCLUSION

Our eventual goal is a system where both data and computation are truly distributed. We envision applications where only a subvolume of a larger dataset is required for a computation, and where that subvolume may be available from various combinations of different sources. We must be able to rapidly identify the set of partial replicas that intersect with the required subvolume, and then choose a specific combination of replicas to read from, so that the performance could be maximized. We develop an SRLS that can rapidly identify the set of intersecting replicas in a distributed environment. In addition, we design and implement a selection model that can choose a combination of replicas that provides strong performance.

We describe a fast replica selection algorithm that provides load balancing and takes file storage organization, filesystem prefetching and hardware performance into account using separate models. We expect the system would benefit various distributed scientific applications in distributed computing systems.

There are several avenues for future research. In the MAQR-tree implementation, we may continue our optimization efforts by replacing the database with our own storage module, giving greater control over disk behavior. Secondly, we entertain the possibility of using GPUs to not only support user computation directly, but to also assist in computation intensive selection and management tasks.

In our replica selection, adding more storage models should allow us to accommodate a wider variety of datasets. We may replace our static network model with a grid *Monitoring and Discovery System (MDS)* [69] to address fluctuating network performance. Also, we can improve our device model to eliminate the error when a query falls upon the jump edge shown in figure

5.12(b). Most importantly, we must also address the more general replica selection scenario in which replicas may overlap in an arbitrary way, adding considerable complexity and requiring the special properties of an R-tree or similar data structure.

Although our SRLS and data server support queries from multiple clients simultaneously, the Replica Selection component is currently not able to process selection requests from multiple clients simultaneously. In a real world grids or cloud, the bandwidth and latencies, computation load and resources are always dynamically changing. In this case, the subsequent selection solution might be influenced by a previous selection, which constitutes a more complicated problem. In the future, we will address this interesting problem by coupling our system with real time Monitoring Services, and ensure that the current scheduling decision takes previously scheduled network, server, and disk load into account.

In the future, we will extend our system to incorporate both the unstructured GIS datasets and the array-based datasets on one platform. For instance, during climate simulation, a climatologist attempts to find an optimal and safe route for valley residents to evacuate. The simulation involves array-based climate simulation data and unstructured or structured GIS data. We envision a future system where the climate simulation and resulting data is fully distributed with accuracy determined by the needs of the climatologists, rather than available local capacity. We expect our system to provide very useful supports for this type of computation.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] S. Vazhkudai, S. Tuecke, and I. Foster, "Replica selection in the globus data grid," in *Proc. CCGRID '01*. Published by the IEEE Computer Society, 2001, p. 106.

[2] Y. Zhao and Y. Hu, "GRESS–a grid replica selection service," in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS-2003)*, 2003.

[3] O. Tatebe, K. Hiraga, and N. Soda, "Gfarm grid file system," *New Generation Computing*, vol. 28, no. 3, pp. 257–275, 2010.

[4] A. Chervenak, R. Schuler, M. Ripeanu, A. Amer, S. Bharathi, I. Foster, A. Iamnitchi, and C. Kesselman, "The globus replica location service: design and experience," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1260–1272, 2009.

[5] Y. Lin, Y. Chen, G. Wang, and B. Deng, "Rigel: A scalable and lightweight replica selection service for replicated distributed file system," in *Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on*. IEEE, 2010, pp. 581–582.

[6] S. Ramakrishnan and P. Rhodes, "Multidimensional replica selection in the data grid," in *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*. IEEE, 2006, pp. 373–374.

[7] L. Weng, U. Catalyurek, T. Kurc, G. Agrawal, and J. Saltz, "Using space and attribute partitioned partial replicas for data subsetting and aggregation queries," in *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 2006, pp. 271–280.

[8] R. Chang and P. Chen, "Complete and fragmented replica selection and retrieval in Data Grids," *Future Generation Computer Systems*, vol. 23, no. 4, pp. 536–546, 2007.

[9] I. F. Haddad, "Pvfs: A parallel virtual file system for linux clusters," *Linux J.*, vol. 2000, no. 80es, Nov. 2000.

[10] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 2:1–2:17.

[11] P. Schwan, "Lustre: Building a file system for 1000-node clusters," in *Proceedings of the 2003 Linux Symposium*, vol. 2003, 2003.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[13] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[14] OpenStack.org, "OpenStack Open Source Cloud Computing Software," 2013. [Online]. Available: http://www.openstack.org/

[15] S. Muraki, E. B. Lum, K.-L. Ma, M. Ogata, and X. Liu, "A pc cluster system for simultaneous interactive volumetric modeling and visualization," in *Parallel and Large-Data Visualization and Graphics, 2003. PVG 2003. IEEE Symposium on*. IEEE, 2003, pp. 95–102.

[16] W. T. Correa, "New techniques for out-of-core visualization of large datasets," Ph.D. dissertation, Princeton, NJ, USA, 2004, aAI3110226.

[17] NASA, "data.NASA an Open NASA Project," 2011. [Online]. Available: http://data.nasa.gov/about/

[18] ——, "NASA and Data.gov," 2012. [Online]. Available: http://www.nasa.gov/open/plan/data-gov.html

[19] A. Chervenak, E. Deelman, I. Foster, L. Guy, W. Hoschek, A. Iamnitchi, C. Kesselman, P. Kunszt, M. Ripeanu, B. Schwartzkopf, H. Stockinger, K. Stockinger, and B. Tierney, "Giggle: a framework for constructing scalable replica location services," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Baltimore, Maryland: IEEE Computer Society Press, 2002, pp. 1–17.

[20] A. Chervenak, N. Palavalli, S. Bharathi, C. Kesselman, and R. Schwartzkopf, "Performance and scalability of a replica location service," *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pp. 182–191, June 2004.

[21] M. Cai, A. Chervenak, and M. Frank, "A peer-to-peer replica location service based on a distributed hash table," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2004, p. 56.

[22] Globus.org, "Globus Toolkit 5.0.3 RLS User's Guide." [Online]. Available: http://www.globus.org/toolkit/docs/5.0/5.0.3/data/rls/user/

[23] W. Peng-fei, F. Yu, C. Bin, and W. Xi, "GloSDC: A Framework for a Global Spatial Data Catalog," in *Geoscience and Remote Sensing Symposium, 2008. IGARSS 2008. IEEE International*, vol. 2. IEEE, 2009.

[24] Y. Wei, L. Di, B. Zhao, G. Liao, A. Chen, Y. Bai, and Y. Liu, "The design and implementation of a grid-enabled catalogue service," in *International Geoscience and Remote Sensing Symposium*, vol. 6, 2005, p. 4224.

[25] L. Di, A. Chen, W. Yang, Y. Liu, Y. Wei, P. Mehrotra, C. Hu, and D. Williams, "The development of a geospatial data grid by integrating OGC web services with globus-based grid technology," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 14, pp. 1617–1635, 2008.

[26] R. Rahman, R. Alhajj, and K. Barker, "Replica selection strategies in data grid," *Journal of Parallel and Distributed Computing*, vol. 68, no. 12, pp. 1561–1574, 2008.

[27] K. Li, H. Wang, K. Cheng, and T. Wu, "Strategies Toward Optimal Access to File Replicas in Data Grid Environments," *Journal of Information Science and Engineering*, vol. 25, no. 3, pp. 747–762, 2009.

[28] S. Narayanan, U. Catalyurek, T. Kurc, V. Kumar, and J. Saltz, "A runtime framework for partial replication and its application for on-demand data exploration," in *High Performance Computing Symposium (HPC 2005), SCS Spring Simulation Multiconference*, 2005.

[29] B. Yan, "Idea—an api for parallel computing with large spatial datasets," Ph.D. dissertation, University, MS, USA, 2009, aAI3385879.

[30] P. J. Rhodes, X. Tang, R. D. Bergeron, and T. M. Sparr, "Iteration Aware Prefetching for Large Multidimensional Scientific Datasets," in *SSDBM'2005: Proc. of the 17th international conference on Scientific and statistical database management*. Berkeley, CA, US: Lawrence Berkeley Laboratory, 2005, pp. 45–54.

[31] P. J. Rhodes and S. Ramakrishnan, "Iteration Aware Prefetching for Remote Data Access," in *Proc. 1st International Conference on e-Science and Grid Computing*, H. Stockinger, R. Buyya, and R. Perrott, Eds., 2005, pp. 279–286.

[32] Y. Gu and R. Grossman, "Udt: An application level transport protocol for grid computing," in *Second International Workshop on Protocols for Fast Long-Distance Networks*, 2003.

[33] R. Grossman, Y. Gu, X. Hong, A. Antony, J. Blom, F. Dijkstra, and C. de Laat, "Teraflows over gigabit wans with udt," *Future Generation Computer Systems*, vol. 21, no. 4, pp. 501–513, 2005.

[34] Y. Gu and R. Grossman, "UDT: UDP-based data transfer for high-speed wide area networks," *Comput. Netw.*, vol. 51, no. 7, pp. 1777–1799, 2007.

[35] udt.sourceforge.net, "UDT: Breaking the Data Transfer Bottleneck," 2011. [Online]. Available: http://udt.sourceforge.net/index.html

[36] Globus.org, "Globus Toolkit Data Management: Key Concepts," 2011. [Online]. Available: http://www.globus.org/toolkit/docs/4.0/data/key/

[37] coastalemergency.org, "Coastal Emergency Risks Assessment." [Online]. Available: http://coastalemergency.org/

[38] Globus.org, "Globus Toolkit Homepage." [Online]. Available: http://www.globus.org/toolkit/

[39] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The Globus striped GridFTP framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2005, pp. 54–64.

[40] B. Schnitzer and S. Leutenegger, "Master-client R-trees: A new parallel R-tree architecture," in *Proceedings of the 11th International conference on Scientific and statistical database management (SSDBM)*. Published by the IEEE Computer Society, 1999, pp. 68–77.

[41] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," in *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1984, pp. 47–57.

[42] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," *ACM SIGMOD Record*, vol. 19, no. 2, pp. 322–331, 1990.

[43] I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *Proceedings of the International conference on Very Large Databases*, 1994, pp. 500–509.

[44] L. Arge, M. Berg, H. Haverkort, and K. Yi, "The priority R-tree: A practically efficient and worst-case optimal R-tree," *ACM Transactions on Algorithms (TALG)*, vol. 4, no. 1, pp. 1–30, 2008.

[45] I. Kamel and C. Faloutsos, "Parallel R-trees," in *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*. ACM, 1992, pp. 195–204.

[46] L. Mutenda and M. Kitsuregawa, "Parallel r-tree spatial join for a shared-nothing architecture," *dante*, pp. 423–430, 1999.

[47] A. Mondal, Y. Lifu, and M. Kitsuregawa, "P2pr-tree: An r-tree-based spatial index for peer-to-peer environments," *Current Trends in Database Technology-EDBT 2004 Workshops*, pp. 516–525, 2004.

[48] M. Feller, I. Foster, and S. Martin, "GT4 GRAM: A functionality and performance study," in *Proc. TeraGrid Conference*, 2007.

[49] S. Leutenegger, M. Lopez, and J. Edgington, "STR: A simple and efficient algorithm for R-tree packing," in *Data Engineering, 1997. Proceedings. 13th International Conference on*, 1997, pp. 497–506.

[50] Globus.org, "Replica Location Service RPC Protocol Description," 2003. [Online]. Available: http://www.globus.org/toolkit/docs/5.0/5.0.3/data/rls/developer/rpcprotocol.pdf

[51] Y. Tian and P. J. Rhodes, "The Globus Toolkit R-tree for Partial Spatial Replica Selection," in *Proceedings of the 2010 11th IEEE/ACM International Conference on Grid Computing*. Brussels, Belgium: IEEE, Oct. 2010, pp. 169–176.

[52] C. Faloutsos and S. Roseman, "Fractals for secondary key retrieval," in *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, 1989, pp. 247–252.

[53] J. Lawder and P. King, "Using state diagrams for hilbert curve mappings," *International journal of computer mathematics*, vol. 78, no. 3, pp. 327–342, 2001.

[54] J. Hellerstein, J. Naughton, and A. Pfeffer, "Generalized search trees for database systems," *Readings in database systems*, p. 101, 1998.

[55] PostGIS EWKB/EWKT Formats. [Online]. Available: http://postgis.refractions.net/documentation/manual-1.5/ch04.html

[56] Y. Tian and P. Rhodes, "A fast location service for partial spatial replicas," in *2011 12th IEEE/ACM International Conference on Grid Computing*. IEEE, 2011, pp. 190–197.

[57] S. Sarawagi and M. Stonebraker, "Efficient organization of large multidimensional arrays," in *International Conference on Data Engineering*, 1994, pp. 328–344.

[58] *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Co., 1966.

[59] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE Trans. Knowledge and Data Engineering*, vol. 13, no. 1, pp. 124–141, 2001.

[60] P. Havlak and K. Kennedy, "An implementation of interprocedural bounded regular section analysis," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 2, no. 3, pp. 350–360, 1991.

[61] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *Computer*, vol. 27, no. 3, pp. 17–28, 1994.

[62] S. Schlosser, J. Schindler, S. Papadomanolakis, M. Shao, A. Ailamaki, C. Faloutsos, and G. Ganger, "On multidimensional data and modern disks," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005, pp. 225–238.

[63] D. Kotz, S. Toh, and S. Radhakrishnan, "A detailed simulation model of the hp 97560 disk drive," Dartmouth College Technical Report PCS-TR94, Tech. Rep., 1994.

[64] E. Shriver, A. Merchant, and J. Wilkes, "An analytic behavior model for disk drives with readahead caches and request reordering," in *Proc. 1998 ACM SIGMETRICS Joint International Conference on Measurement and modeling of computer systems*. ACM, 1998, pp. 182–191.

[65] E. Shriver, C. Small, and K. Smith, "Why does file system prefetching work," in *Proceedings of the 1999 USENIX Annual Technical Conference*, vol. 27, 1999.

[66] L. Weng, U. Catalyurek, T. Kurc, G. Agrawal, and J. Saltz, "Servicing range queries on multidimensional datasets with partial replicas," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2. IEEE, 2005, pp. 726–733.

[67] D. Papadias, "Java implementation of R*-tree." [Online]. Available: http://www.rtreeportal.org/code/Rstar-java.zip

[68] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, pp. 509–517, September 1975.

[69] J. Schopf, M. D'arcy, N. Miller, L. Pearlman, I. Foster, and C. Kesselman, "Monitoring and discovery in a web services framework: Functionality and performance of the globus toolkit's mds4," Argonne National Laboratory, Preprint ANL/MCS-P1248-0405, Tech. Rep., 2005.

[70] I. Foster and C. Kesselman, "Globus: A metacomputing infrastructure toolkit," *International Journal of High Performance Computing Applications*, vol. 11, no. 2, pp. 115–130, 1997.

[71] ——, "The Globus project: A status report," in *Heterogeneous Computing Workshop, 1998.(HCW 98) Proceedings. 1998 Seventh*, 1998, pp. 4–18.

[72] S. Brunett, K. Czajkowski, S. Fitzgerald, C. Kesselman, I. Foster, S. Tuecke, A. Johnson, and J. Leigh, "Application experiences with the Globus toolkit," in *hpdc*. Published by the IEEE Computer Society, 1998, pp. 81–88.

[73] I. Foster, "Globus toolkit version 4: Software for service-oriented systems," *Network and parallel computing*, pp. 2–13, 2005.

[74] Globus.org, "Globus Toolkit Web Site." [Online]. Available: http://www.globus.org/toolkit/

[75] S. Krishnamurthy, W. H. Sanders, and M. Cukier, "A dynamic replica selection algorithm for tolerating timing faults," in *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*. IEEE, 2001, pp. 107–116.

[76] Z. Lu and K. S. McKinley, "Partial replica selection based on relevance for information retrieval," in *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 1999, pp. 97–104.

[77] C. Ferdean and M. Makpangou, "A scalable replica selection strategy based on flexible contracts," in *Internet Applications. WIAPP 2003. Proceedings. The Third IEEE Workshop on*, 2003, pp. 95–99.

[78] S. Mikami, K. Ohta, and O. Tatebe, "Using the gfarm file system as a posix compatible storage platform for hadoop mapreduce applications," in *Grid Computing (GRID), 2011 12th IEEE/ACM International Conference on*, 2011, pp. 181–189.

[79] P. Wu, Y. Fang, B. Chen, M. Yan, and Y. Zhao, "The Replica Management in a Spatial Data Grid," in *Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference on*. IEEE, 2009, pp. 1–5.

[80] PostGIS, "PostGIS Webpage." [Online]. Available: http://postgis.refractions.net/

[81] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, *Rtrees: theory and applications*. Springer, 2006.

[82] S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee, "Performance evaluation of main-memory r-tree variants," in *Advances in Spatial and Temporal Databases*. Springer, 2003, pp. 10–27.

[83] L. Arge, M. De Berg, H. J. Haverkort, and K. Yi, "The priority r-tree: a practically efficient and worst-case optimal r-tree," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004, pp. 347–358.

[84] R. Bayer, "The universal b-tree for multidimensional indexing: General concepts," in *Worldwide Computing and Its Applications*. Springer, 1997, pp. 198–209.

[85] F. Ramsak, V. Markl, R. Fenk, M. Zirkel, K. Elhardt, and R. Bayer, "Integrating the ub-tree into a database system kernel," in *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, Eds. Morgan Kaufmann, 2000, pp. 263–272.

[86] M. Krátkỳ, J. Pokornỳ, and V. Snášel, "Indexing xml data with ub-trees," *Proceedings of Advances in Databases and Information Systems, ADBIS*, pp. 155–164, 2002.

[87] R. Fenk, A. Kawakami, V. Markl, R. Bayer, and S. Osaki, "Bulk loading a data warehouse built upon a ub-tree," in *Proceedings of the 2000 International Symposium on Database Engineering & Applications*. IEEE Computer Society, 2000, pp. 179–187.

[88] D. Meagher, "Geometric modeling using octree encoding," *Computer graphics and image processing*, vol. 19, no. 2, pp. 129–147, 1982.

[89] H. Samet, "The quadtree and related hierarchical data structures," *ACM Computing Surveys (CSUR)*, vol. 16, no. 2, pp. 187–260, 1984.

[90] G. J. Sullivan and R. L. Baker, "Efficient quadtree coding of images and video," *Image Processing, IEEE Transactions on*, vol. 3, no. 3, pp. 327–331, 1994.

[91] F. Losasso, F. Gibou, and R. Fedkiw, "Simulating water and smoke with an octree data structure," in *ACM Transactions on Graphics (TOG)*, vol. 23, no. 3. ACM, 2004, pp. 457–462.

[92] K. Zhou, Q. Hou, R. Wang, and B. Guo, "Real-time kd-tree construction on graphics hardware," in *ACM Transactions on Graphics (TOG)*, vol. 27, no. 5. ACM, 2008, p. 126.

[93] S. Berchtold, D. Keim, and H. Kriegel, "The X-tree: An index structure for high-dimensional data," *Readings in multimedia computing and networking*, vol. 12, p. 451, 2002.

[94] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Computing Surveys (CSUR)*, vol. 30, no. 2, pp. 170–231, 1998.

[95] Y. Fang, M. Friedman, G. Nair, M. Rys, and A. Schmid, "Spatial indexing in microsoft SQL server 2008," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM, 2008, pp. 1207–1216.

[96] P. Ganesan, B. Yang, and H. Garcia-Molina, "One torus to rule them all: multi-dimensional queries in p2p systems," in *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, ser. WebDB '04. New York, NY, USA: ACM, 2004, pp. 19–24.

[97] C. Luk and T. Mowry, "Compiler-based prefetching for recursive data structures," in *ACM SIGOPS Operating Systems Review*, vol. 30, no. 5. ACM, 1996, pp. 222–233.

[98] ——, "Automatic compiler-inserted prefetching for pointer-based applications," *IEEE Transactions on Computers*, vol. 48, no. 2, pp. 134–141, 1999.

[99] J. Rao and K. Ross, "Making b+-trees cache conscious in main memory," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 475–486.

[100] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Informed prefetching and caching," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*. ACM, 1995, pp. 79–95.

[101] S. Chen, P. Gibbons, and T. Mowry, *Improving index performance through prefetching*. ACM, 2001, vol. 30, no. 2.

[102] S. Chen, P. Gibbons, T. Mowry, and G. Valentin, "Fractal prefetching b+-trees: Optimizing both cache and disk performance," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002, pp. 157–168.

[103] H. Kang, J. Kim, D. Kim, and K. Han, "An Extended R-Tree Indexing Method Using Selective Prefetching in Main Memory," *Computational Science–ICCS 2007*, pp. 692–699, 2007.

[104] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "Fast: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the 2010 international conference on Management of data*. ACM, 2010, pp. 339–350.

[105] P. Lindstrom, "Out-of-core construction and visualization of multiresolution surfaces," in *Proceedings of the 2003 symposium on Interactive 3D graphics*, ser. I3D '03. New York, NY, USA: ACM, 2003, pp. 93–102.

[106] P. Sulatycke and K. Ghose, "A fast multithreaded out-of-core visualization technique," in *Parallel and Distributed Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*. IEEE, 2002, pp. 569–575.

[107] S. Narayanan, T. Kurc, U. Catalyurek, and J. Saltz, "Database support for data-driven scientific applications in the grid," *PPL-Parallel Processing Letters*, vol. 13, no. 2, pp. 245–272, 2003.

[108] H. Song, Y. Yin, Y. Chen, and X. Sun, "A cost-intelligent application-specific data layout scheme for parallel file systems," in *Proceedings of the 20th international symposium on High performance distributed computing*. ACM, 2011, pp. 37–48.

VITA

Mr. Yun Tian worked for three years in the China National Railway Signal & Communication Corp. from 1999 to 2002. During that time, he worked on many projects to test train and subway automatic control systems. Mr. Yun Tian received his Bachelor of Science in Computer Information Management in 2006 from Fudan University and his Master degree of Science in Applied Computer Technology from Shanghai Normal University in 2008.

At the University of Mississippi, as a research assistant, Mr. Tian have worked with Dr. Philip Rhodes on Partial Replica Location and Selection for Spatial Datasets. Meanwhile, he taught three courses as a graduate instructor, including Survey of Computing, Parallel Programming and Programming for Science and Engineering. In addition, he volunteered at many events, such as local Lego League Robot Competition and Regional Mathcounts Competition. He also volunteered for one semester as a Lab Assistant for the class of Programming for Science and Engineering.

His research interests include large scientific data in distributed computing, replica selection, distributed and parallel programming, GPGPU computing, High-Performance Computing, data transfer services, spatial databases and spatial indexing methods and compression on GPUs.

Mr. Tian received many honors and awards, including Outstanding Student, Honors Fellowship and Research Fellowship etc. He depicts himself as a person with great passion and a tender loving heart.