GRIMS: A Scalable Management and Storage System for Massive Remote Sensing Images

Liang ZHAO[#1], Luo CHEN[#2], Ning JING[#3], Huaiyu ZUO[#4]
[#]College of Electronic Science and Engineering
National Univ. of Defense Technology,
Changsha 410073, China
[1]zlbusiniao@gmail.com

ABSTRACT

Remote sensing images are important in ecological, geographical and military applications. With the rapid growing volume of remote sensing images, how to manage and store the massive remote sensing images is becoming a must-be-solved problem. We build a scalable storage and management system for massive remote sensing images aiming to store global remote sensing images – Global Remote Sensing Images Management and Storage system (GRIMS). In GRIMS, we propose a tile pyramid model – Plate Carree Projection Grid Quad-Tree (PCPGQT) to spatially partition the high resolution images and build a double tower (DT) index for the big image data file. Meanwhile, we analyze the fragment problem in image mosaic. Through fragment collection, huge disk spaces could be saved. By using the open source software HADOOP, we realize a distributed storage system to store massive image data file. Experiments show that our tile pyramid model is suitable to support our system purpose, and the distributed storage system is highly efficient.

Keywords

RS images, GRIMS, Management, Partition, Storage, D-MaRISS

1. INTRODUCTION

Remote sensing(RS) technologies, combined with geographical information systems(GIS) , have been applied into various fields such as government services, agriculture, oil and mineral exploration, emergency services, environmental monitoring, land using, urban planning, and ecological research[1]. Remote sensing data is the most important data for geographical analyzing. However, Earth scientists have been thwarted by the staggering volume of remote sensing images in their attempts to study the earth; either the numbers or the quality of various remote sensing platforms is greatly increasing. Facing the large volumes of data received from remote sensing sensors, how to manage and store the massive remote sensing images is becoming a must-be-solved problem.

The characteristics of the remote sensing database are [2]: massive, spatial referenced, multi-spectrum, multi-platform, rich information, etc. Thus, a system for remote sensing images must meet the following requirements [3]:

Spatial Query-Large Image databases must be highly efficient in supporting spatial query. So an efficient spatial index and the corresponding processing mechanism are needed.

Multi-Resolution Support-This is driven by two factors: high cost of operations on full-resolution raster data; access restrictions to the full-resolution data.

Mass Storage-Large image databases require storage capacities in the range of hundreds of gigabytes to terabytes.

High-Performance Storage-Image databases with simultaneous multi-user access additionally demand data transfer bandwidths exceeding the I/O performance of individual hard disks.

Considering the characteristics of remote sensing images and the system requirements, we design and implement a distributed management and storage system for massive remote sensing images—GRIMS (Global Remote Sensing Images Management and Storage system). Global means that our system is scalable to store global massive remote sensing images. The logic architecture of GRIMS is showed in figure 1.

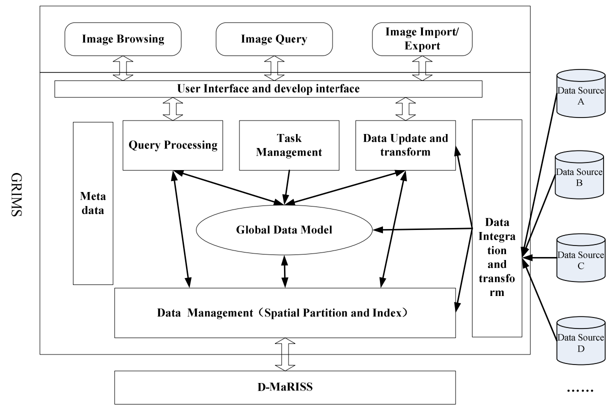


Figure 1. Logic Architecture of GRIMS.

GRIMS is composed of several main parts: "Global Data Model", "Task Management", "Query Processing", "Data Update and Transform", "Meta data", "User Interface and Develop Interface", "Data Integration and Transform", "Data Management", "D-MaRISS (Distributed-Massive Remote-sensing Image Storage System)". The last two components: "Data Management" and "D-MaRISS" are the cores of the whole system.

"Data Management" module deals with spatial partition and index. Based on the map projection theory, we propose a new tile pyramid model for spatial partition, build the corresponding index file. Meanwhile, we present a tile-based mosaic method to ensure the seamless mosaic of massive images. What's more, we discuss the fragment collection problem, thus huge disk spaces could be saved.

“D-MaRISS” module is the distributed storage system for massive remote sensing images. We discuss the design principles for distributed storage of remote sensing images. Based on the open source software HADOOP, we realize a distributed massive remote sensing images storage system and explain the mechanism for storage.

Research on the above two modules and the key technologies in them are the main contributions of this paper.

The rest of the paper is organized as follows: section 2 is about related work; section 3 discusses the management of remote sensing images; section 4 involves with the distributed storage of remote sensing images; section 5 is about experiments and analysis; we make our conclusions in section 6.

2. RELATED WORK

Researchers have done superior works in order to efficiently store, organize, manage and publish massive remote sensing images. Their works can be mainly divided into two categories:

The first category is: massive remote sensing images storage and management systems based on traditional RDBMS. By using the traditional RDBMS, there exist two technique routes: One is to spatially extend the RDBMS in order that it possesses the ability to manage raster type remote sensing images. Various Database Companies are the main driven force in this route, and Oracle Spatial GeoRaster[4][5][6] is the typical product. Another route to provide the ability of storage and management of massive remote sensing images is to implement spatial data engine (SDE) which acts as a middleware between applications and RDBMS. Most of the SDEs are implemented by GIS companies, among which ArcSDE[7] developed by ESRI is the most typical product. Compare the two technique routes, when using the former technique; we use extended SQL or database packages to access images in the database. Due to lack of consistent standards, different RDBMSs provide quite different access interfaces. However, by using SDEs, we use object-oriented style APIs, thus SDEs are ORM[8](object relation mapping) middle wares for spatial databases.

From its early start in 1970s till now, traditional RDBMS technology is quite mature, but is still lack of efficiencies in the following aspects when trying to store and process remote sensing images: rapid access ability to massive data, flexibility, processing unstructured data, cost of storage and maintenance, rapid backup and recovery. Some of the shortcomings are essentially caused by the relational model of the traditional RDBMS. What's more, generic commercial DBMS is huge in scale, which leads to great system cost and difficulties when being deployed. All the above reasons become obstacles in using RDBMS as the foundation of system for massive remote sensing images.

The second category is: massive remote sensing images storage and management systems based on generic massive storage systems. Network storage technology characterized by NAS(Network Area Storage) and SAN(Storage Area Network) is becoming more and more popular. HPSS[9][10] (High Performance Storage System, developed by IBM, US Department of Resource, Laboratory of Lawrence Livermore), RASCHAL[11][12](developed by NASA Jet Propulsion Laboratory, JPL), Blue Whale[13][14](developed by Chinese academy of sciences) all use this technologies. Among them, RASCHAL which is being used in the OnEarth[15] project by NASA JPL, stores more than 15TB global image mosaics. Moreover, Blue Whale has been used in many systems for remote sensing images in China.

Generic storage systems concentrate on storage capacity and access speed, while aiming at more storage and fast reading and writing speed. It doesn't care the key issues that a system for the massive remote sensing images focuses on: effectively organize the image data, construct the indexes for image data, and leverage the cluster's parallel computing ability and data compression algorithms. The idea: to buy a commercial massive storage system, import the image data and gain an efficient storage and management system for massive remote sensing images is naive and unrealistic.

Besides the above two categories, online high resolution remote sensing images browsing systems are flourishing. Google Maps and Google Earth are the outstanding systems among them. According to reference[16] published in 2005, remote sensing data that support the Google Maps/Earth come to 70.5TB, among which images account for 70TB and index data accounts for 500GB. From 2005 till now, Google must have updated their remote sensing data, so the up-to-date data should exceed 100TB. Google implements many original key technologies in building its large scale data center, including: Google Cluster[17], Workqueue[18], GFS[19], MapReduce[20], Bigtable[16], Sawzall[18], Chubby[21].

Considering the excellent online service quality provided by Google Maps/Earth, we reckon that Google's experiences in storage and management for remote sensing images are representative of the research trend, thus are valuable to our work.

3. MANAGEMENT OF REMOTE SENSING IMAGES

The size of individual remote sensing images and image mosaics often exceeds the amount of primary memory available. Typically, only parts of these objects are accessed at any given time and operations affecting the entire object are relatively rare. A spatial massive images management solution should therefore support the transparent spatial partitioning and reconstruction of large image objects. The efficient access to partitions or to individual pixels or cells of large spatial image objects requires the use of specialized spatial access or indexing methods.

3.1 Spatial Partition and Index

Spatial Partition aims to divide high resolution images into low resolution small tiles and build the index for each tile. GRIMS incorporates a spatial partition concept based on tile pyramid model.

The tile pyramid model in our system involves with two classical map projection methods in map projection theory--Plate Carree projection and Mercator projection. Map projection, which projects the original images onto a datum plane, is a key step in the pre-processing of remote sensing images. By using the rectified images, the tile pyramid model partitions the global images into tiles, extracts pyramids at different levels and builds the corresponding indices. The two projection methods are popularly used in the storage systems of remote sensing images. Plate Carree projection is used in BMNG[22] and OnEarth, while

Mercator projection is used in Google Maps, Microsoft Live Maps[23] and Yahoo Maps[24].

In Mercator projection, showed in figure 2, images that range from longitude -180 to 180 and latitude -85.05112877980659 to 85.05112877980659 are projected onto a regular square plane. The discrete multi-resolution pyramids that are extracted based upon that square plane can be quad-partitioned at each level and thus can be indexed by using quad-tree. We name the tile pyramid model based on Mercator projection--MPQT (Mercator Projection Quad-Tree). But as it can not cover the global images (lack of latitude coverage), it is not a suitable model for global images management. Left of figure 2 shows the projected global images, right of figure 2 shows its code string.

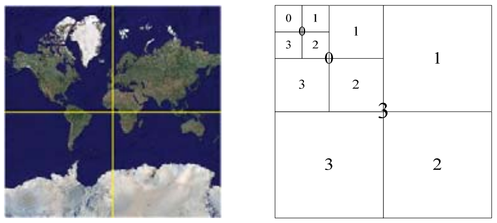


Figure 2. MPQT Partition.

While in Plate Carree projection, showed in figure 3, images that range from longitude -180 to 180 and latitude -90 to 90 are projected onto a rectangle plane which maintains a length and height ratio 2:1. The discrete multi-resolution pyramids that are extracted based on that rectangle plane can be uniformly partitioned at each level and thus can be indexed by using grid. We name the tile pyramid model based on Plate Carree projection -- PCPG (Plate Carree Projection Grid). In PCPG, tiles at each level are indexed by its column and row number. When indexing the 30 level tiles, the integer type can not hold such big number and the index file could be so huge.

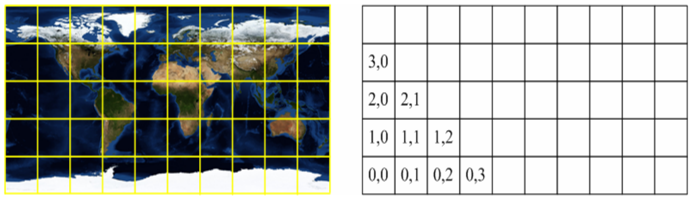


Figure 3. PCPG Partition.

Based on the two models, we propose a new model named PCPGQT (Plate Carree Projection Grid Quad-Tree). In PCPGQT, we use the Plate Carree projection together with the quad-tree indexing to each tile at different levels.

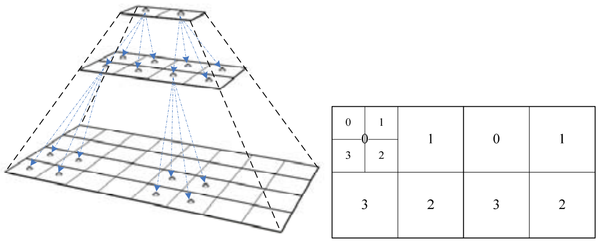


Figure 4. PCPGQT Partition.

As the left of figure 4 shows, the top of the pyramid is a 1*2 grid. Take the top level two tiles as the root node, extend downwards, two quad-trees can be formed. In our model, the structure of the two quad-trees is called Double Towers (DT) index. Every tile has a unique code string. Let *n* denote the node in the two quad-trees except the root node. *f(n)* is the parent node of n, *C(n)* is the code string. First, we define the Azimuth code:

[Definition 3.1 Azimuth Code]: We partition a plane uniformly into several sub areas and assign each sub area with a unique code. These codes are call Azimuth code.

Specially, if this partition can be executed recursively, the Azimuth code of the sub areas in every partition result is invariable. For PCPGQT, in the two quad-trees of DT, the square area is partitioned uniformly by using 0, 1, 2, 3 as the Azimuth code (as the right of figure 4 shows). In every recursion the four sub areas are coded clockwise from the northwest. For every node *n*, let *c(n)* be its Azimuth code, *R(c)= {0, 1, 2, 3}*. Based on the definition of the Azimuth code, the code string to node n can be constructed according to the following rules:

- For the top two tiles in the pyramid, their *C(n)* are 'w' and 'e', with each presenting the western and eastern hemisphere separately.
- Coding for the nodes in the two quad-trees are the same (the two quad-trees are called w-tree and e-tree).
- For each non-root node *n* in w-tree (or e-tree), we have *C(n)=C(f(n))+c(n)*, in which '+' is defined as string appending operation.

Based on the above coding rules, we can use the longitude and latitude value and pyramid level to get the code string *C(n)* for each node *n*. The algorithm can be described as follows:

```
GetTileCodeString(lon, lat, level)
Initialize minx, maxx, miny, maxy, nodestring
if lon ≥ 0.0
    nodestring ← nodestring + 'e'
    minx ← 0.0
else
    nodestring ← nodestring + 'w'
    maxx ← 0.0
for i ← 1 to level
    middlex ← (minx + maxx) / 2.0
    middley ← (miny + maxy) / 2.0
    if lon < middlex and lat ≥ middley
        nodestring ← nodestring + '0'
        maxx ← middlex
        miny ← middley
    if lon ≥ middlex and lat ≥ middley
        nodestring ← nodestring + '1'
        minx ← middlex
        miny ← middley
    if lon ≥ middlex and lat < middley
        nodestring ← nodestring + '2'
        minx ← middlex
        maxy ← middley
    if lon < middlex and lat < middley
        nodestring ← nodestring + '3'
        maxx ← middlex
        maxy ← middley
end for
return codestring
```

Figure 5. GetTileCodeString Algorithm.

The computing complexity is *O(level)*.

We implement the popular two models and compare them with our proposed model in section 5.

3.2 Tile-based Mosaic and fragment collection

Through spatial partition, we get a lot of tiles and their corresponding index codes. The next problem to solve is what should be put into database and how are they organized? This problem is discussed in the following sub sections.

3.2.1 *Data Preparing*

In GRIMS, four parts of data should be put into database.

1. Image data file

Tiles are the basic read and write unit in our system. We have two ways to serialize them into the file system: tile as a single file; all tiles are organized into a big file. For the first method, we can use the file name as the index key, while for the second method, we have to define and maintain the index key by ourselves. When the images are massive, the file system has to manage huge amount of files, for example, when the images go to TB amount, the number of tiles will be so big, and thus, access to such massive tiles would be a big burden for the file system. On the other hand, there is no such problem by using the second method. So, we use the second method to serialize our massive images.

2. The index file

We implements three different tile pyramid models in our system. MPQT uses pyramid quad-tree index, PCPG uses pyramid grid index, PCPGQT uses DT index.

Figure 6 shows the structure of the DT index file. DT index file is actually a two-branch quad-tree. The root node stores two separate parts with each part containing a character representing the w-tree or the e-tree and a pointer to its child node. The tree node maintains four parts. Every part in the node contains a character representing the Azimuth code. Moreover, it also contains two pointers: one pointer points to the child node, while the other one points to the image tile in the data file.

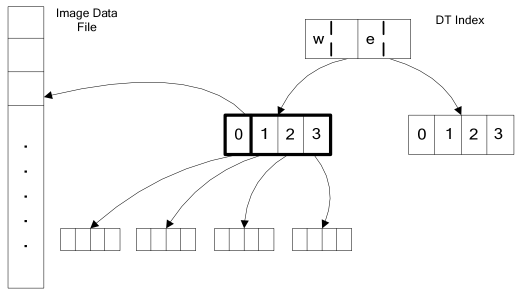


Figure 6. Structure of the DT index file.

3. The Metadata file

With reference to the standard of remote sensing metadata published by the Federal Geographic Data Committee (FGDC) [25], we build our own metadata file. In our metadata file we record the following contents: the sensor's information, the frequency information, the spatial reference, the tile size, the pyramid level and the data compression method. The metadata file is serialized into a XML file.

4. The additional information file

The fragment list is described in section 3.2.3.

3.2.2 *Tile-Based Mosaic*

We can ensure the exact level of each image in the pyramid through the longitude and latitude vale. But the size of the image generally are not integral times of the partition grid (we use 512*512 grid). In order to describe this situation, we first propose some definitions:

[Definition 3.2 Minimal Grid Bounding Rectangle (MGBR)]: the minimal rectangle that an image takes up at some level in the pyramid is called its MGBR.

[Definition 3.3 Natural Tile]: the image tile that is fully covered by the image in its MGBR.

[Definition 3.4 Inferior Tile]: the image tile that is partially covered by the image in its MGBR.

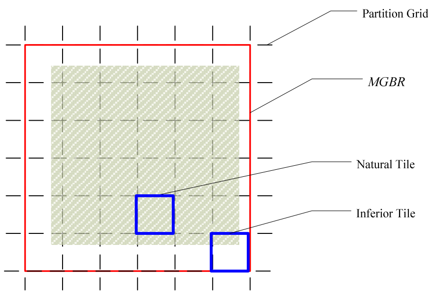


Figure 7. Tiles in the partition grid.

Figure 7 shows the natural tiles and inferior tiles in the MGBR. The uncovered part in the tile is generally filled by black point in our system.

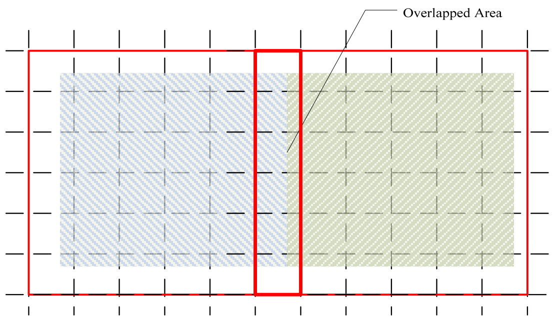


Figure 8. Overlapped Area.

It is a typical and common situation in the mosaic operation that two MGBRs are overlapped at some inferior tiles, as figure 8 shows. To solve this problem and thus make the whole mosaic seamless, we define a tile-based mosaic (TIM) operation. It means that when we mosaic two inferior tiles, the value of the pixels in the overlapped area are calculated by '|' (or) operation. And so, we call this a "Tile-based Mosaic" method.

3.2.3 *Fragment Collection*

TIM operation frequently occurs in the partition process, this TIM operation thus will produce a lot of fragments. A fragment means the storage space in the data file that can not be reassigned or reused. Figure 9 shows the process that how fragments are produced.

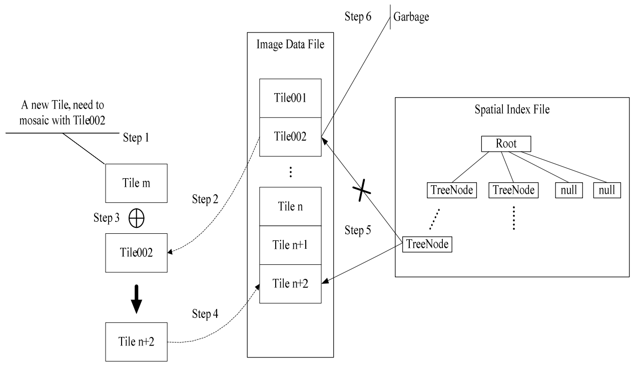


Figure 9. How fragments are produced.

When a new tile m is being written to the data file:

- Step1: Judge whether Tile m is overlapped with the existing tiles. Now, we assume that Tile m is overlapped with Tile 002, so a TIM operation is needed.
- Step2: Extract the data of Tile 002 from the data file by using the spatial index file.
- Step3: Execute TIM operation and we get a new tile named Tile n+2. Because in our system we use data compression algorithm (different pixel values get different compression results), the storage space for every tile maybe different. The new Tile n+2 generally is larger than the original Tile 002, so we can not insert the new tile into the original space.
- Step4: Append Tile n+2 to the end of the data file.
- Step5: Update the spatial index; change the pointer of Tile 002 to the new Tile n+2.

Through the above 5 steps, the original space that holds the data of Tile 002 becomes a fragment. Our experiment shows, because of the frequent TIM operations, the fragments can be up to 50%~60% in the data file. To solve this waste of disk space, we propose a "Fragment Collection" method.

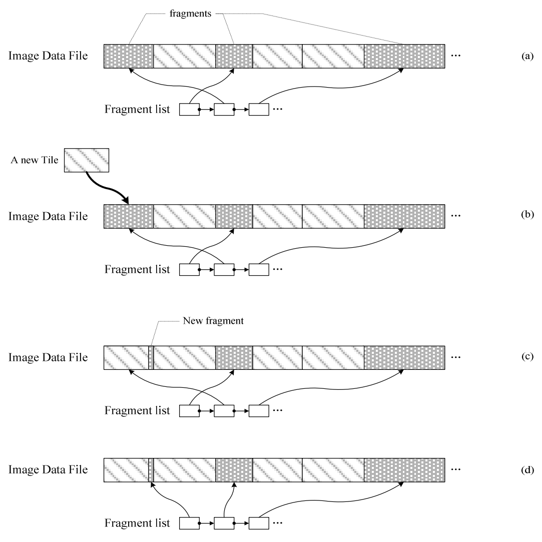


Figure 10. Fragment collection process.

As figure 10 shows, the fragment collection process can be described as the following steps:

- Step1: Initialize a fragment information list (FL) to record the position and size of all the fragments. The FL is an ordered linked-list, the nodes in the list are ordered from small to big by the size of the fragment it points to. (Figure (a) shows)
- Step2: When a new tile is being written to the data file, we first check the FL to find whether there exists a fragment that can hold the tile. If there is more than one suitable fragment, we choose the smallest and put the tile data in it. (Figure (b) shows)
- Step3: Record the remnant space as a new fragment and insert into FL. (Figure (c) shows)
- Step4: Update the FL to make sure that the fragments are ordered from small to big. (Figure (d) shows)
- Step5: If all the fragments can not hold the tile, we append the new tile to the end of the data file.

Although by using this method, we can not diminish all the fragments, but we can indeed decrease the fragment in the data file and thus greatly improve the usage of disk space.

4. DISTRIBUTED STORAGE OF REMOTE SENSING IMAGES

The research to meet the rapidly growing demands of massive remote sensing images presents two trends: one is the architecture of storage changes from centralized to distributed and from small scale clusters to large scale clusters; the second is data management based on traditional RDBMS changes to specified systems that consider the special properties of remote sensing images.

Using the distributed file systems to store massive remote sensing data has the following two merits: the cost is low because no commercial DBMS is needed; the performance can be optimized through operations to file system at operating system level. So according to analysis in section 2 and the above reasons, we choose the distributed file system to store massive remote sensing images.

At present, we could use a lot of distributed file systems like AFS, NFS, Lustre, GPFS, which can meet the needs for general purpose applications. But when trying to use them as the infrastructure for specified applications of massive remote sensing images, we could find that they are lack of efficiencies.

According to Google's GFS and based on the open source software HADOOP, We have designed and implemented the D-MaRISS (Distributed-Massive Remote-sensing Image Storage System）to meet the rapidly growing demands of storing massive remote sensing images. D-MaRISS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our technological environment, which reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

- The system stores a number of large image data files. Multi-GB files are the common case and should be managed efficiently.

- The system is built from many inexpensive commodity components typically Linux computers in our system.
- The workloads have many large, sequential writes that append data to files. Typical operation sizes are similar to those for reads. Once written, files are seldom modified again. Small writes at arbitrary positions in a file are supported but do not have to be efficient.
- The system must efficiently implement well-defined semantics for multiple clients.

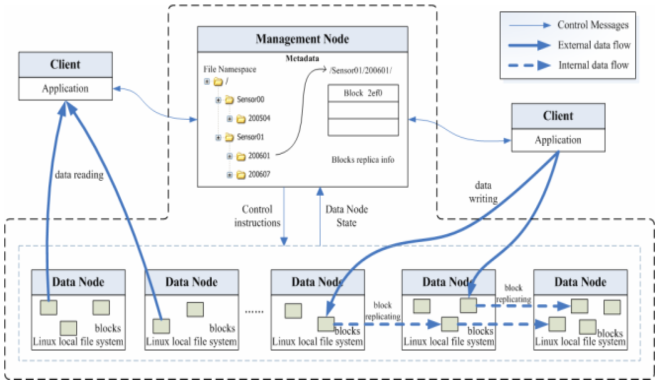


Figure 11. The architecture of D-MaRISS.

A D-MaRISS cluster consists of a single management node and multiple data nodes and is accessed by multiple clients, as showed in figure 11. Each of these nodes is typically a commodity Linux machine running a user-level server process. It is easy to run both a data node and a client on the same machine, as long as machine resources permit and the lower reliability caused by running possibly flaky application code is acceptable.

Files are divided into fixed-size blocks. Each block is identified by an immutable and globally unique 64 bit block handle assigned by the management node at the time of block creation. Data nodes store blocks on local disks as Linux files and read or write block data specified by a block handle and byte range. For reliability, each block is replicated on multiple data nodes. By default, we store three replicas, though users can designate different replication levels for different regions of the file namespace.

With reference to Figure 11, let us explain the interactions for a simple read. First, using the fixed block size, the client translates the file name and byte offset specified by the application into a block index within the file. Then, it sends the management node a request containing the file name and block index. The management node replies with the corresponding block handle and locations of the replicas. The client caches this information using the file name and block index as the key. The client then sends a request to one of the replicas, most likely the closest one. The request specifies the block handle and a byte range within that block. Further reads of the same block require no more client-management node interaction until the cached information expires or the file is reopened. In fact, the client typically asks for multiple blocks in the same request and the management node can also include the information for blocks immediately following those requested. This extra information sidesteps several future client-management node interactions at practically no extra cost.

In GRIMS, the data file is stored in D-MaRISS. The other files mentioned above are stored in the client computers.

5. EXPERIMENTS AND ANYLISIS

We do some experiments to test our methodology; the environment can be described as follows:

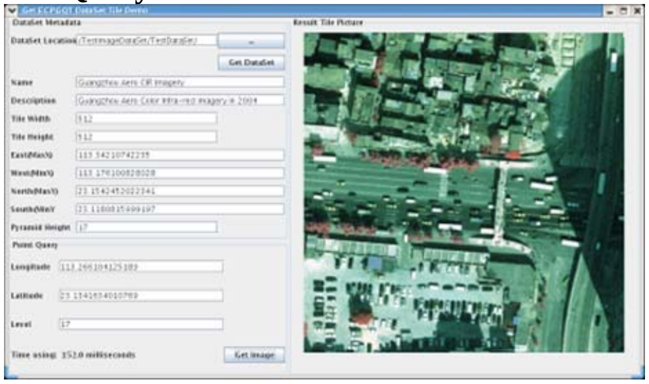
- Clusters: We maintain a local network of 1 management node, 10 data nodes, 12 clients through a Gigabit Ethernet switcher. Every node has the same configurations, as figure 12 shows:

CPU:	1×Celeron 3.2GHz
Memory:	2×1GB
Hard Disk:	80GB SATA-II 7200RPM
Ethernet Card:	100Mbps
Operating System:	RHEL 5
Linux Local File System:	XFS 1.4

Figure 12. Configurations of every node in the cluster.

- Distributed File System: We use the Hadoop Distributed File System (HDFS) module provided by the open source software HADOOP (version 0.12.0). The number of replications to each block is 3 and the block size is 64M.

5.1 Query Performance



DataSet Metadata
DataSet Location /TestImageDataSet/TestDataSet/
Name Guangzhou Aero CIR Imagery
Description Guangzhou Aero Color Infra-red imagery in 2004
Tile Width 512
Tile Height 512
East(MaxX) 113.34210742235
West(MinX) 113.176100828028
North(MaxY) 23.1542452022341
South(MinY) 23.1160015999197
Pyramid Height 17
Point Query
Longitude 113.266104125183
Latitude 23.1341654010789
Level 17
Time using: 152.0 milliseconds

Figure 13. Query and browse interface.

Figure 13 is the query and browse interface. We carry out our experiments based on it. The query process begins with designating an image data set. Then it shows some metadata of the data set. With user input query value, the requested image could be fetched. And the time used could be displayed.

We realize the three abovementioned spatial partition model in our system and test the cost of the spatial index and the query performance. The image data set used in this experiment has the following properties: 22.3GB, 0.2 meter resolution, infrared aeronautic remote sensing images.

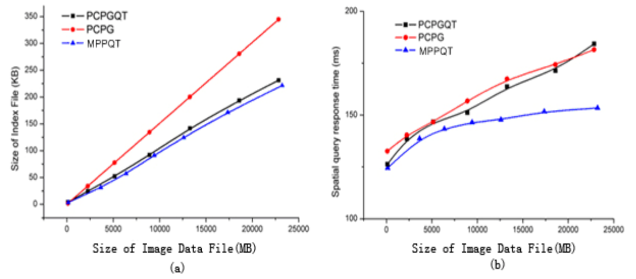


Figure 14. Size of index file and query performance.

Figure 14 (a) shows that the size of the spatial index file to the size of the remote sensing images data file is 1:100000. So, we can infer that the size of the index file of the PB amount images is 10 to 100 GB. In a large scale distributed storage environment, this index file can by stored directly in the memory to support highly efficient query. Figure 14 (b) shows that the difference between the query performances based on different model is not notable. With the growing size of data file, the change of the query performance is not obvious either. All the queries could be completed in 200 ms.

From figure 14 we can see that our proposed model PCPGQT is not the best either in size of the index file or in the query performance. But as the MPPQT model can not deal with global remote sensing images, what's more, the PCPG model is weaker than PCPGQT in either the two aspects. So we still choose our PCPGQT as the spatial partition model. It is the best tradeoff between our aim and performance.

5.2 Throughput of D-MaRISS

The second experiment involves with testing the throughput of D-MaRISS. For a node connected to a 100Mbps Ethernet card, its theoretical saturated transfer rate (STR) is 12.5MB/s. But, the average actual transfer rate (AATR) we obtain in our test is about 11.219MB/s which account for 89.75% of the theoretical value. In our experiment, we test the concurrent read or write performance of D-MaRISS through increasing the clients, thus the AWR or ARR can be recorded.

We write a program based on socket and read/write interfaces provided by the local file systems to test the average actual transfer rate (AATR). Meanwhile, we directly invoke the read and write functions provided by D-MaRISS to get the ARR and AWR values.

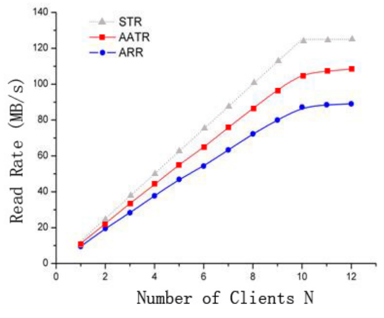


Figure 15. Aggregate read rate.

We have a 16GB image data set which contains 32 files which are 512MB. The data set is stored in D-MaRISS. N clients read randomly from the 32 files. Figure 15 shows the test result for ARR test. The top curve shows theoretical limits imposed by our network topology which is accord with formula (1). (M denotes the number of data nodes; N denotes the number of clients). The middle curve shows the aggregate read rate that we could get in our own experiment environment. The bottom curve shows the ARR. As we can see, the ARR increases with the numbers of client. When the client number exceeds 10, either STR, AATT or ARR all become steady with little change. The STR becomes a fixed value 125MB/s after the number of clients goes to 10. This is because we maintain 10 data nodes in the cluster.

$$\begin{cases} MAX(STR_{read}) = N \cdot STR_{DataNode} & (0 \le N \le M) \\ MAX(STR_{read}) = M \cdot STR_{DataNode} & (N \ge M) \end{cases} \quad (1)$$

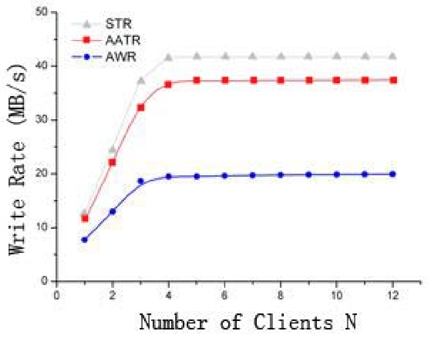


Figure 16. Aggregate write rate.

For AWR test, every client has a 1GB image file to write to D-MaRISS. Figure 16 shows the result for AWR test. The write rates are much lower than the write rates because D-MaRISS maintains 3 replications of the original file. The STR goes to its max value 41.7MB/s after the number of clients goes to 10. Its value is accord with formula (2).

$$\begin{cases} MAX(STR_{write}) = N \cdot STR_{DataNode} & (0 \le N \le M) \\ MAX(STR_{write}) = \dfrac{M \cdot STR_{DataNode}}{REPLICAS} & (N \ge M) \end{cases} \quad (2)$$

Experiments by Google in ref [19] show similar results in throughput test.

6. CONCLUSIONS

In order to overcome the growing gap between the production rate of remote sensing images and the almost complete absence of suitable management solutions, we build a highly efficient and scalable storage and management system for remote sensing images. The primary objective of GRIMS is to store global remote sensing images. For this purpose, we implement our system upon distributed file system to make it scalable; meanwhile we design a suitable partition model to organize the global remote sensing images. The key elements of our solution can be summarized as follows:

- Tile pyramid-based image partitioning
- DT index

- Fragment collection
- Spatial tile query
- Highly scalable storage system upon ordinary Linux desktops

The system tests demonstrated a good scalability and performance. Our proposed method and system can meet the needs of storage and management of massive remote sensing images mentioned in section 1. So, GRIMS is a scalable and global-oriented system.

But, right now, we haven't considered the temporal aspect of remote sensing images, so it is our way ahead.

7. ACKNOWLEDGMENTS

The authors would like to pay our sincere thanks to Lu LIU who is now in Tech Univ of Munich, and Guo CHENG. Only with their talented work can we finish this paper. Also, thanks should be paid to the excellent programmers in Google. Our idea is originated from their famous products. Last, thanks for the support by NSF 40601080.

8. REFERENCES

[1] Shuo Dong, Qiaoli Hu. 2005 Building remote sensing database on Grid. IGARSS-Volume 1 (July 25-29, 2005).

[2] Shock, C., Chang, C., Davis, L., Goward, S., Saltz, J. H., and Sussman, A.1996. A High Performance Image Database System for Remotely Sensed Imagery. In Proceedings of the Second international Euro-Par Conference on Parallel Processing-Volume II (August 26 - 29, 1996). Lecture Notes in Computer Science, vol. 1124. Springer-Verlag, London, 109-122.

[3] Stephan NEBIKER, 1998 Girds-An Architecture for Managing Very Large Orthoimage Mosaics in A Database Framework. IAPRS.

[4] Oracle Corp. Oracle Spatial 11g GeoRaster. Oracle Technical White Paper, 2007.7.

[5] Oracle Corp. Oracle Spatial GeoRaster 10g Release 2 (10.2) User's Guide and Reference (B14254-01). 2005.6.

[6] Oracle Corp. Oracle Database 10g GeoRaster: Scalability and Performance Analysis. Oracle Technical White Paper, 2005.8.

[7] ESRI ArcSDE. http://www.esri.com/software/arcgis/arcsde/

[8] Scott W. Ambler. Mapping Objects to Relational Databases: O/R Mapping In Detail. http://www.agiledata.org/essays/mappingObjects.html

[9] HPSS: High performance storage system, http://www.sdsc.edu/hpss/hpss.html

[10] Richard W. Watson. High Performance Storage System Scalability: Architecture, Implementation and Experience. Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies, MSST 2005. pp. 145-159.

[11] RASCHAL: Raid Again Storage using Commodity Hardware And Linux, http://pat.jpl.nasa.gov/public/lucian/RASCHAL.html, last updated: 2005.12.18

[12] Lucian Plesea. Remote Access to Very Large Image Repositories, 2005. A High Performance Computing Perspective. Pasadena, CA, USA: Jet Propulsion Laboratory, NASA, 2005.

[13] Yang Dezhi, Huang Hua, Zhang Jiangang, Xu Lu. 2005. BWFS：A Distributed File System with Large Capacity, High Throughput and High Scalability. Journal of Computer Research and Develop, 2005.6. 42(6).

[14] Huang Hua, Zhang Jiangang, Xu Lu, 2005. Distributed Layered Resource Management Model in Blue W hale Distributed File System. Journal of Computer Research and Develop, 2005.6. 42(6).

[15] NASA JPL OnEarth, http://onearth.jpl.nasa.gov/index.html

[16] Chang, F., Dean, J., Ghemaw, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. 2006. Bigtable: A distributed structured data storage system. In 7th OSDI (2006). pp. 205–218.

[17] Luiz André Barroso, Jeffrey Dean, Urs Holzle. 2003. The Google Cluster Architecture. USA: IEEE Computer Society, 2003.

[18] Rob Pike, Sean Dorward, Robert Griesemer, and Sean Quinlan. 2005. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, 13(4):277–298, 2005.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, 2003. The Google File System, Proc. 19th Symposium on Operating System Principles, Lake George, New York, 2003, pp. 29-43.

[20] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proc 6th Symposium on Operating Systems Design and Implementation, San Francisco, 2004, pages 137-149.

[21] BURROWS, M. 2006. The Chubby lock service for loosely coupled distributed systems. In Proc. of the 7th OSDI (Nov. 2006).

[22] Reto Stockli, Eric Vermote, Nazmi Saleous, Robert Simmon and David Herring, 2005. The Blue Marble Next Generation - A true color earth dataset including seasonal dynamics from MODIS, http://snowy.arsc.alaska.edu/nasa/bmng.pdf , October 17, 2005.

[23] Microsoft Live Maps (Microsoft Virtual Earth), http://maps.live.com

[24] Yahoo Maps, http://maps.yahoo.com

[25] FGDC, Content Standard for Digital Geospatial Metadata: Extensions for Remote Sensing Metadata. http://www.fgdc.gov/standards/projects/FGDC-standards-projects/csdgm_rs_ex/MetadataRemoteSensingExtens.pd