International Journal of *Geo-Information*

Article

An Effective NoSQL-Based Vector Map Tile Management Approach

Lin Wan [1], Zhou Huang [2,*] and Xia Peng [3,4]

1 Faculty of Information Engineering, China University of Geosciences, Wuhan 430074, China; wanlin@cug.edu.cn
2 Institute of Remote Sensing & GIS, Peking University, Beijing 100871, China
3 Institute of Tourism, Beijing Union University, Beijing 100101, China; ivy_px@163.com
4 State Key Laboratory of Resources and Environmental Information System, Institute of Geographical Sciences and Natural Resources Research, Chinese Academy of Sciences, Beijing 100101, China
* Correspondence: huangzhou@pku.edu.cn; Tel.: +86-10-6276-0132

Academic Editors: Bert Veenendaal, Maria Antonia Brovelli, Serena Coetzee, Peter Mooney and Wolfgang Kainz
Received: 15 August 2016; Accepted: 5 November 2016; Published: 12 November 2016

Abstract: Within a digital map service environment, the rapid growth of Spatial Big-Data is driving new requirements for effective mechanisms for massive online vector map tile processing. The emergence of Not Only SQL (NoSQL) databases has resulted in a new data storage and management model for scalable spatial data deployments and fast tracking. They better suit the scenario of high-volume, low-latency network map services than traditional standalone high-performance computer (HPC) or relational databases. In this paper, we propose a flexible storage framework that provides feasible methods for tiled map data parallel clipping and retrieval operations within a distributed NoSQL database environment. We illustrate the parallel vector tile generation and querying algorithms with the MapReduce programming model. Three different processing approaches, including local caching, distributed file storage, and the NoSQL-based method, are compared by analyzing the concurrent load and calculation time. An online geological vector tile map service prototype was developed to embed our processing framework in the China Geological Survey Information Grid. Experimental results show that our NoSQL-based parallel tile management framework can support applications that process huge volumes of vector tile data and improve performance of the tiled map service.

Keywords: digital map; map tile; NoSQL; MapReduce; cloud computing

1. Introduction

Digital maps, or online electronic maps, are the core components of modern Web geographic information systems (GIS). Supported by various thematic information databases, web-based map series can be provided with different scales for multiple geographic themes, such as streets, traffic, politics, and agricultural phenomena. The web map service framework can be considered as a computing framework for the integration of hardware, algorithms, software, networking and spatial data. With the rapid growth of various networked geospatial data sources, such as volunteered geographic information in social networks, real-time data acquired from sensor networks and large quantities of earth observation data from satellites, web-based electronic map service infrastructures have faced the big challenge of organization and processing strategies for the huge volume of spatial data and its inherent complexity. This has resulted in the pursuit of advanced high-performance computing architectures and sophisticated algorithms to achieve scalable map data storage and fast processing.

Currently, most map service systems, such as Google Maps, Yahoo! Maps, Bing Maps, and OpenStreetMap, are primarily built on vector tile map models, in which a set of map images that are pre-generated or dynamically rendered from vector GIS data are delivered to the requester via a web browser. To facilitate visualization in different resolutions and map-based spatial analysis, the entire vector map should be split into many chunks with specific clip algorithms before tile generation. The map tiles, or so-called tile caches, are basic units of one specific tile level and are identified by the corresponding zoom level, row and column. There are many tile storage structures, and the Tile Pyramid structure based on a level-of-detail (LOD) model is renowned for its efficiency for large map image rendering and fast tile scheduling. The use of the Tile Pyramid model can support the display of high-resolution tile images with a high level of performance.

The sizes of tiles in Tile Pyramid structures are growing quickly, and the number of users with concurrent access is also increasing because geo-social apps are used by more and more people. However, the traditional tile storage and management strategies, standalone high-performance computer (HPC) architectures and commercial relational databases cannot easily accomplish the goals of preprocessing, integration, exploration, scheduling, and visualization for big-data tiles. When there is a sharp rise in tile data volume and the number of concurrent sessions, these classic management models exhibit characteristics of slow reading and writing speed, limited storage capacity, poor scalability and high building and maintenance cost. Hence, within such a big geo-data environment, the development of effective mechanisms which facilitate fast tile operations becomes a key requirement. Technical issues should be addressed to implement the combination of distributed vector tile data and distributed geoprocessing by means of a software system.

We concentrate more on the exploration of massive vector tile data management methods, particularly on a versatile scalable computing framework that provides both distributed tile storage and parallel vector tile processing (tile generation and fast retrieval). The framework forms the link between the network map data organization model and the spatial operations that are associated with this distributed model. In recent studies, cloud-based mechanisms, such as Not Only SQL (NoSQL) storage and the MapReduce programming paradigm, have been gradually adopted by many researchers in the geocomputing field to explore new methods for large-volume spatial data processing. The number of cloud geocomputing frameworks has proliferated, and they are varied in their main points of studies. For example, some researchers emphasized distributed storage means for tile storage, whereas other studies focused on MapReduce-based spatial data processing against one specific batch-oriented, high-latency platform such as Apache Hadoop. With the need for a systematic approach to satisfy both mass storage and fast parallel processing for big tile data, how to integrate a scalable map data model with fast operating capabilities is becoming a critical goal of cloud-based tile computing.

The objectives of this paper focus on two aspects. First, we propose a new tile data storage model based on one type of NoSQL paradigm: wide column-oriented storage. The model distributes massive tile data across scalable data nodes while maintaining global consistency and allows the effective handling of specific tile data. Second, based on our tile data model, a parallel algorithm framework was presented for fast vector tile generation and querying using the MapReduce programming model. We also incorporate the new algorithm framework into a tiled map service as a component of a real geological survey information vector map service prototype and assess its performance with other methods of our NoSQL tile data infrastructure. In the remainder of the paper, we sequentially introduce applications of the cloud-based computing method, such as Hadoop, NoSQL, and MapReduce, in big geo-data processing. This is followed by an illustration of the extended vector map tile generation and locating algorithm, preparing for the NoSQL-based tile data model. Next, we outline vector map tile storage access models and illustrate how to map tile data into NoSQL column-family stores. The tile map parallel clipping and query framework based on MapReduce is discussed in detail. We also devote a section to the elaboration of the parallel tile generation and static/dynamic query algorithm. After the description of the delivery of the parallel tile operation algorithm as a GIS service, we conduct a series of comparative experiments to evaluate its performance. Finally, we close with some concluding remarks.

2. Related Work

2.1. Cloud-Based Big Geo-Data Processing

Our main work objective is an endeavor to integrate proper cloud computing mechanisms based on open standards with massive spatial information data management, particularly vector map tile data management. In the geospatial domain, some progress has been made towards implementing an operational big geo-data computing architecture. Part of the research has focused on web service-based geoprocessing models for distributed spatial data sharing and computing [1–4], and some studies have investigated CyberGIS-based [5,6] methods to address computationally intensive and collaborative geographic problems by exploiting HPC infrastructure, such as computational grids and parallel clusters [7–9], particularly focused upon the integration of particular CyberGIS components, spatial middleware and high-performance computing and communication resources for geospatial research [10,11].

Cloud computing is becoming an increasingly more prominent solution for complex big geo-data computing. It provides scalable storage to manage and organize continuously increasing geospatial data, elastically changing its processing capability to data parallel computing with an effective paradigm to integrate many heterogeneous geocomputing resources [12,13]. Thus far, the advancements of cloud-enabled geocomputing involve dealing with the intensities of data, computation, concurrent access, and spatiotemporal patterns [14]. To date, the integration of advanced big data processing methods and practical geocomputing scenarios is still in its infancy and is a very active research field.

2.2. Hadoop and Big Geo-Data Processing

Hadoop is a reputable software ecosystem for distributed massive data processing in a cloud environment. As an open-source variant of Google's big data processing platform, its primary aim is to provide scalable storage capability with lower hardware costs, effective integration of any heterogeneous computing resources for cooperative computing, and a highly available cloud service on top of loosely coupled clusters.

As the core component, Hadoop Distributed File System (HDFS), which runs on scalable clusters consisting of cheap commodity hardware, can afford reliable large-dataset (typical in the GB to TB range) storage capabilities with high bandwidth and streaming file system data access characteristics. Based on this, it could accelerate intensive computing on massive data based on the idea that "moving computation is cheaper than moving data." By adopting a master/slave architecture, the master node of HDFS, often called NameNode, maintains the global file metadata and is responsible for data distribution as well as accessing specific data blocks, whereas slaves—DataNodes, also commonly called computing nodes—manage their own data region allocated by the master node, performing data block creation, modification, deletion and replication operations issued from the NameNode. On the basis of these advanced distributed file system features, HDFS meets the requirement for large-scale and high-performance concurrent access and has gradually become a foundation of cloud computing for big data storage and management.

Another core component of Hadoop is the MapReduce [15] parallel data processing framework. Built on a Hadoop cluster, the MapReduce model can divide a sequential processing task into two sub-task execution phases—Map and Reduce. In the Map stage, one client-specified computation is applied to each record unit (identified by unique key/value pairs) of the input dataset. The execution course is composed of multiple instances of the computation, and these instance programs are executed in parallel. The output of this computation—all intermediate values—is immediately aggregated (merged by the same key identifier) by another client-specified computation in the Reduce stage. One of the most significant advantages of MapReduce is that it provides an abstraction that hides many system-level details from the client. Unlike other parallel computing frameworks, the MapReduce model can decompose complex algorithms into a sequence of MapReduce parallel jobs without considering their communication process.

Hadoop MapReduce is the open-source implementation of the programming model and execution framework built on HDFS. In this framework, the job submission node (typically NameNode) runs the JobTracker, which is the administration point of contact for clients wishing to execute a parallel MapReduce job. The input to a MapReduce job comes from data stored on the distributed file system or existing MPP (massively parallel processing) relational databases, and the output is written back to the distributed file system or relational databases; any other system that satisfies the proper abstractions can serve as a data source or sink. In the Map stage, JobTracker receives MapReduce programs consisting of code segments for Mappers and Reducers with data configuration parameters, divides the input into smaller sub-problems by generating intermediate key/value pairs, and distributes them to each TaskTracker node (usually DataNodes) in the cluster to solve sub-problems, whereas the Reduce procedure merges all intermediate values associated with the same key and returns the final results to the JobTracker node. During the entire process, JobTracker also monitors the progress of the running MapReduce jobs and is responsible for coordinating the execution of the Mappers and Reducers. To date, besides Hadoop, there are also many other applications and implementations of the MapReduce paradigm, e.g., targeted specifically for multi-core processors, for GPUs (Graphics Processing Unit), and for the NoSQL database architecture.

Because Hadoop provides a distributed file system and a scalable computation framework by partitioning computation processes across many computing nodes, it is better for addressing big data challenges and has been widely adopted in big geo-data processing and analysis scenarios [16–18].

2.3. Not Only SQL (NoSQL)-Enabled Big Data Management

The emergence of NoSQL () databases was accompanied by the urgent demand for new methods to handle larger-volume data, which forced the exploration of building massive data processing platforms through distributed clusters of commodity servers. In addition to the underlying processing platform, the data model must be concerned with making upper applications that are more appropriate for fast data storage and parallel computing, for which the relational model has some limitations, such as impedance mismatch [19] and the fact that relational tuples do not support complex value structures, etc. Generally speaking, NoSQL databases refer to a number of recent non-relational databases, which are schema-less and more scalable on clusters and have more flexible mechanisms for trading off traditional consistency for other useful properties, such as increased performance, better scalability and ease of application development, etc.

Currently, there are several primary types of NoSQL databases:

- Key/Value data model: This NoSQL storage is simply considered as one hash table or one relational tuple with two attributes—a key and its corresponding value—and all access to the database is via the primary key. Because key-value storage always uses primary-key access, it generally has great performance and can be easily scaled. The typical implementation of the Key/Value data model is the distributed in-memory databases, such as Memcached DB, Redis, Riak and Berkeley DB.
- Document data model: This NoSQL model stores document contents in the value part of the Key/Value model by XML, JSON and BSON types. It can be recognized as key-value storage, in which the value is self-describing. With a document model, we can quickly query part of the document rather than the entire contents, and data indexes can be created on the contents of the document entity. Some of the popular implementations of the document data model are MongoDB, CouchDB, and RavenDB.
- Column-Family data model: Column-family storage can be considered a generic key-value model. The only special part is that the value consists of multiple column-families, each of which is a

Map data structure. In this Map structure, each column has to be part of a single column family, and the column acts as the basic unit for access with the assumption that data for a particular column family will usually be accessed together. As an open-source clone of Google's Bigtable, HBase is a sparse, distributed, persistent multidimensional implementation of the column-family model, indexed by row key, column key, and timestamp and frequently used as a source of input and as storage of MapReduce output. Other popular column-family databases are Cassandra, Amazon DynamoDB and Hypertable.

- Graph data model: The graph data model focuses on the relationships between the stored entities and is designed to allow us to find meaningful patterns between the entity nodes. The graph data model can represent multilevel relationships between domain entities for things such as category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Some representative examples are Neo4J, FlockDB and OrientDB.

Based on the good scalability, large-scale concurrent processing capability and MapReduce parallel model of NoSQL databases, some preliminary research has been made toward exploring spatial data distributed storage and processing [20–24].

3. Design Issues

With the rapid development of the Internet, the map tile service is more and more widely used, and a large number of users might access the service at the same time. This creates processing pressure of the server, thus using high-performance computing technologies to optimize map tile data management and access becomes the natural choice. Tile data management approaches of current mainstream map tile services, such as Google Maps, Bing Maps, Baidu Map, etc., are still black boxes and not known to the outside world. Therefore, in view of the tremendous advantages of the emerging NoSQL storage and MapReduce parallel computing framework, we try to explore the key technologies of using NoSQL databases to manage the map tile data, including fast tile-clipping methods and real-time tile query approaches. In particular, we use open source NoSQL databases rather than commercial products to achieve efficient management of map tile data. The goal of this paper is to provide a low-cost, effective and transparent NoSQL-based technical framework associated with key approaches for readers who need to build their own map tile services.

The map tile service we have envisioned consists of two types of application scenarios:

(1) The user is interested in a single layer and submit query and visualization requests. In this case, tile data of the single layer are generated into a huge number of pictures stored on the server side in advance. Hence, when the query request arrives, the server will return the corresponding tile picture collection according to the query range. Generating tiles in advance is also called *static tile clipping*.

(2) The user is interested in a number of layers and submit query and visualization requests. In this case, because the user-requested layers cannot be predicted in advance, tile data of multiple layers are dynamically generated based on the Tile Pyramid model and user-defined parameters, only after query requests arrive. At this point, multiple layers are superimposed and rendered together and returned as a set of tile images. Generating tiles in the running time is also called *dynamic clipping*. Furthermore, the dynamically generated clipped tiles are then stored in the database. Hence, when a similar request arrives next time, the result can be returned directly without dynamic clipping.

Corresponding to the above two scenarios, the NoSQL database, with distributed storage and the parallel computing framework (i.e., MapReduce framework), fully meets the needs of map tile clipping and querying. A typical example, as shown in Figure 1, illustrates how distributed storage and MapReduce framework work together in data processing. Firstly, there is a cluster comprising of multiple computer nodes. Then, in the distributed storage level, the data are partitioned into several blocks and stored separately at different nodes. Therefore, when a query or processing request is submitted, several parallel map tasks associated with the data blocks are set up. After the parallel map tasks are finished, the reduce task is performed to obtain the final computing result by merging temporary results output by the map phase. Furthermore, the MapReduce framework sets up TaskTracker on each running node to manage and track the map or reduce task, and JobTracker is built for the whole MapReduce workflow's control issues.

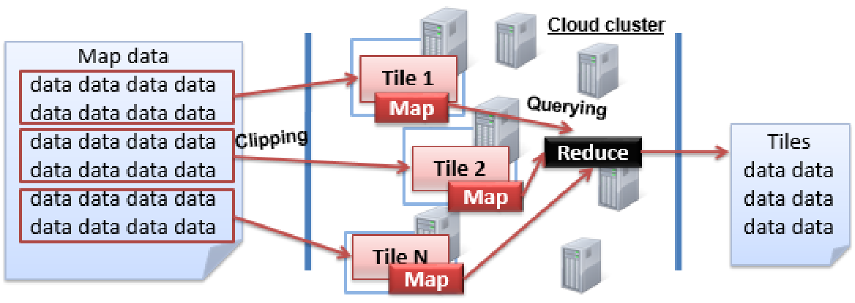

Figure 1. MapReduce framework based map tile data management.

Through the concurrent map tasks, the map data can be quickly clipped into tiles and distributed onto NoSQL storage, and real-time tile queries can also be achieved through the MapReduce framework. In addition, there are four types of NoSQL storage models, including key-value, document, column and graph. The key-value database such as Redis is often memory-based, but the huge amount of map tile data is often much larger than the amount of memory. The graph database is used for graph-type data management. The document database like MongoDB is often used for semi-structured data (e.g., social media user data) management. The column database physically stores the data in accordance with the column, and thus achieves a better data compression rate compared with the row storage. In particular, it is critical to use a column database to physically store the tile data together to improve the I/O efficiency in query processing. Therefore, we use HBase, the most popular open source column database, to store massive map tile data, and HBase's built-in MapReduce framework is utilized to achieve both effective map clipping and real-time tile querying.

Furthermore, as to map tile services, spatially adjacent tiles are often accessed and returned together to the user, so the adjacent tiles with the same level need to be stored in a server or adjacent servers in the HBase's cluster environment as much as possible. This avoids/reduces cross-server data access, thereby improving efficiency. At the same time, under the HBase's column storage model, tiles are also put together in the physical storage level, significantly improving the I/O efficiency, and thus meeting real-time access needs better.

4. Vector-Based Tile Generation and Locating Method

4.1. Basic Methods

Tile maps typically use the Tile Pyramid model with a quadtree tile-index structure (as shown in Figure 2) for data organization. As a top-down hierarchical model of variable resolution, many map service providers, such as Google Maps, Bing Maps and ArcGIS Online, use it to organize their large map tiles.

In the Tile Pyramid model, each *tile size* is fixed to 256 × 256 pixels and *level* determines the map size. At the lowest level of detail (Level 0), the global map is 256 × 256 pixels. When the level increases, the map size, including width and height, grows by a factor of 2 × 2: Level 1 is 512 × 512 pixels, Level 2 is 1024 × 1024 pixels, Level 3 is 2048 × 2048 pixels, and so on. Generally, the map size (in pixels) can be calculated as Equation (1):

$$\text{map width} = \text{map height} = 2^{\text{level}} \times 256 \tag{1}$$

As shown in Figure 1, a vector dataset can be organized into multi-level maps using a quadtree recursive subdivision strategy. Then, each tile (represented by one rectangular block) of one specific level can be clipped and rendered into one picture as a part of the tile map. For a given map tile data, we can use the triple T = {*level*, *row*, *col*} to identify it, where *level* represents the resolution level of the tile, starting with level 0; *row* and *col* denote the row coordinate and column coordinate of the corresponding tile in the current level, respectively. Obviously, the N-level (N > 0) tile layer is the successive result clipped from the N-1 tile level. For instance, tile level 1 is the further clipped result of level 0; the rows and columns of level 0, bisected in half, result in a total of four tiles, and the row coordinates and column coordinates can simply be numbered from 0 to 1. After that, the upper left corner of the tile map can be identified by tile triple {1, 0, 0}. Likewise, level-2 tile generation is based on level-1 tiles, and each tile will be equally divided into four portions, namely, rows and columns were dissected into four equal portions, resulting in a total of 16 tiles. Then, the q-level tile map can have its rows and columns divided equally into 2^q portions, finally resulting in a total of 4^q tiles after rendering operations. Using this method, we can accurately derive the triple identification for any tile of any level.

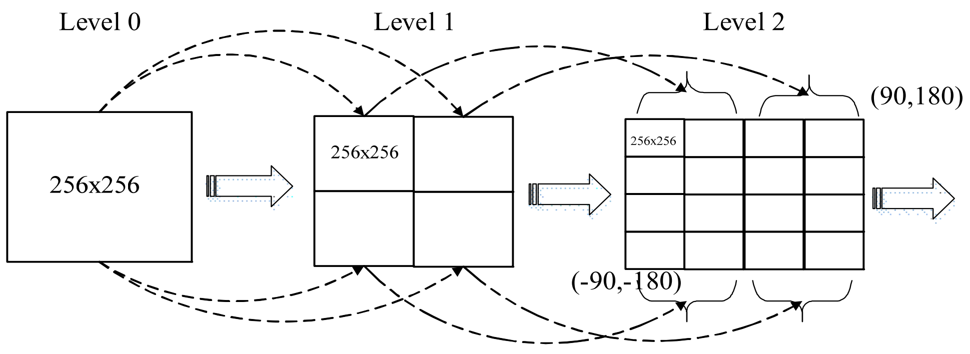

Figure 2. Quadtree-based tile-clipping pattern for global map tile generation.

In the application of a massive tile map service, a tile map with a very large coverage area is often generated from many sub-maps. Hence, different tile partitions clipped by a sub-vector dataset need to be identified in the context of a larger tile map. The key to a map tile-locating algorithm is how to calculate the tile identifier set covered by the query region. Different sub-map documents could have inconsistent ranges, resulting in differences in the row- and column-encoded rule of the tile cache generated by the clipping operation for each map document. To conduct a global uniform tile locating operation in practical applications, we need to specify the origin for both the global tile grid of the large map and the local tile grid of the sub-map. In Figure 3, tile-locating methods for a map of China, used as an example, illustrate the tile retrieval course in a large vector map scope. In this example, before starting to render tile data for the sub-scope, we need first to designate the origin *O* for the global map grid. Here, we set the geographic coordinate of the upper left corner of China's geographic scope to be the origin of the global grid, and we also specify the display ratio for each level and the height and width of the tile unit. Then, both the size of the grid of each specific level and the row and column number covered by the global grid can be determined. The sub-map region in Figure 3 is divided into a grid that has eight rows and ten columns. When some local area in this scope needs to be clipped into tiles, we can quickly find the corresponding grid identifiers at the same level of the global coverage grid. Point *O'* in Figure 3 represents the origin of one sub-map's local area in the global map and is numbered 22 in the country tile grid at the same level. Using this method, we can use geographic coordinates to compute the corresponding tile level and the coverage area in the level for tile query requests and then easily calculate the tile identifiers covered by this geographic range. Vector tiles of large maps, such as the China map example in Figure 3, can be clipped in parallel and deployed in a distributed manner using this tile generation strategy.

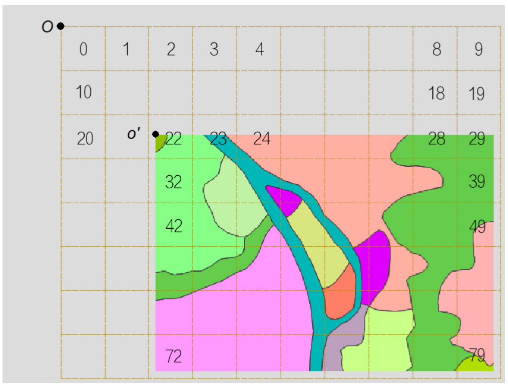

Figure 3. Local map area locating in a global tile map grid.

4.2. Tile-Clipping Range Expansion Algorithm

Because map documents for clipping are often in an irregular figure pattern, generating tiles by the map range will cause the clipping results to be inconsistent with the tile range defined by the Tile Pyramid model. In other words, the original map range needs to be extended to just cover an integer number of titles. Therefore, we present a range expansion algorithm to extend the original range of the map, taking the expanded range as the clipping range.

A range expansion algorithm primarily refers to appropriately adjusting the actual data range with certain rules according to the clip goal to easily address it. Generally, let the irregular clip region expand to a rectangular (rectangle or square) area, resize the zone height and width to the same size as the tile grid unit, and then set the expanded scope as the new clipping range.

For the sake of simplicity, in our study, we extend the map document to a square figure before the tile data generation. This method primarily sets the left bottom as the origin of the original scope, expanding left to right and bottom to top. The original scope is expanded to a new clipping area according to this rule; then, the processing vector data for generating each level's map tiles is increased by the exponential growth model in this context.

Assume that the original range is expressed as $\{(x0_{min}, y0_{min}), (x0_{max}, y0_{max})\}$, the expanded range is denoted as $\{(x_{min}, y_{min}), (x_{max}, y_{max})\}$, and the height and width of tile grid unit are represented by *Height* and *Width*, respectively.

Let $\text{rate0} = (y0_{\text{max}} - y0_{\text{min}})/(x0_{\text{max}} - x0_{\text{min}})$; thus,

(1) When $rate0 > Height/Width$, then $y_{max} = y0_{max}$, $(y0_{max} - y0_{min})/{=}(x_{max} - x0_{min}) = (Height/Width)$, and the tile map expands to the left horizontally, as shown in Figure 4.

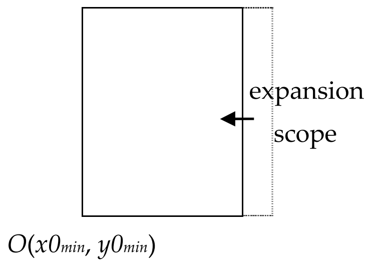

Figure 4. Tile map horizontal expansion.

(2) In Figure 5, if $rate0 = Height/Width$, i.e., $x_{max} = x0_{max}$, $y_{max} = y0_{max}$, then the tile map does not need any expansion.

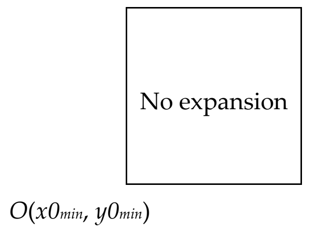

Figure 5. No expansion in the tile map.

(3) If *rate0* < *Height*/*Width*, then $x_{max} = x0_{max}$, $(y_{max} - y0_{min})/(x_{max} - x_{Min}) = Height/Width$, and the tile map expands to the top vertically in Figure 6.

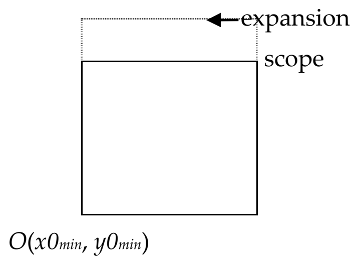

Figure 6. Tile map vertical expansion.

5. NoSQL-Based Vector Map Tile Storage Access Model

A tile map pyramid, as a hierarchical model with variable resolution, has many excellent characteristics, such as massive storage capacity and a dynamic update mechanism. Using the world vector tile map as an example, the total tile number of the 1 to 20 zoom level is nearly 3×10^{12}. In this massive data scenario, the relational database and file system-based management approach, which is not suitable for the storage and processing of numerous tiles, will seriously affect data storage and retrieval actions, with slow computation and update rate. NoSQL databases support mass data storage and highly concurrent queries with better scalability and other great features as well. Moreover, some NoSQL variants, such as HBase, building on top of a distributed file system, also provide a fault-tolerant, replica and scalable storage mechanism and are suitable for processing huge quantities of tile data records. Therefore, we here consider applying NoSQL methods to build a distributed storage management model for massive tile map data.

5.1. Distributed Storage Model Based on Column-Family

In our studies, map tile data, which are clipped from a vector dataset, are stored using a NoSQL column-family model similar to Google's BigTable. In HBase, a *table* is comprised of a series of rows consisting of a *RowKey* and a number of *ColumnFamilies*. *RowKey* is the primary key of the table, and the records in the table are sorted according to *RowKey*. A *ColumnFamily* can be composed of any number of *Columns*, that is, *ColumnFamily* supports dynamic expansion without the need to pre-define the number and type of *Column*.

<*map name, level, row, column, reverse timestamp*> is an option for RowKey design, but in this case the row key is a string with variable-length and delimiters should be used to connect the separated elements. One alternative is to use the hash algorithm like MD5 to obtain a fixed-length row key. Using "*MD5 (Map_Name) MD5 (Level) MD5 (Row) MD5 (Col) + reverse timestamp*" as RowKey, there are two advantages: (1) fixed-length RowKey could help HBase better predict the read and write performance; (2) RowKey is usually expected to be evenly distributed; the MD5-based method enables this goal. Because data blocks are automatically distributed onto region servers by row keys in HBase, evenly

distributed keys can avoid the problem of writing hot spots, and thus avoid the load concentration in a small part of the region servers.

As shown in Figure 7, *RowKey* is used to globally identify a specific tile object, initially composed of a map document identifier, tile level, row number, and column number. Furthermore, to facilitate frequent updating and fast locating, a reverse timestamp for per data update actions is also considered as one component of *RowKey* to ensure that the latest version of the updated tile data is automatically arranged at the top of the tile data table. This allows the latest tile version to be given the utmost access priority in a query scheduling session. Based on that, we use the method of calculating 16-bit MD5 (Message-Digest Algorithm 5) values of each field and then combine them together to yield the *RowKey* (i.e., *GMK* function as shown in Figure 8). In column-family storage, we adopt the single column-family structure—only one ColumnFamily, *Tile* and one Column, *Image* to speed up data access while avoiding the wide-column performance cost. The value of *Image* is the corresponding map binary data for this tile unit.

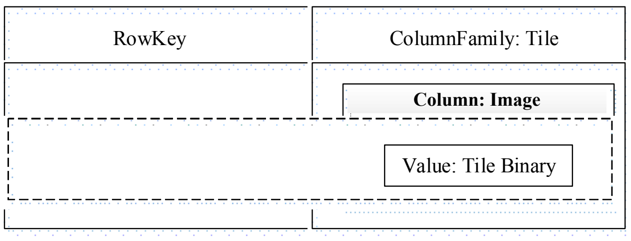

Figure 7. Column-family-based tile data storage model.

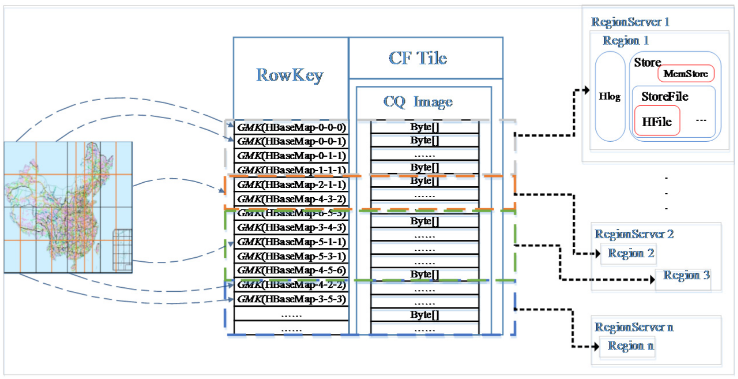

Figure 8. Distribution of map tiles in a NoSQL cluster.

Based on this, we propose a NoSQL cluster-based map tile distributed storage model: as shown in Figure 8, in the process of clipping, the different levels of tile data of the map document generate the corresponding calculated *RowKey*, namely, the tile identifier, according to the *GMK* method (i.e., "*MD5 (Map_Name) MD5 (Level) MD5 (Row) MD5 (Col) + reverse timestamp*"). Then, tile data blocks are stored into records of the NoSQL column table, and different records are assigned to different *Region* nodes according to the partitioning strategy; here, *Region* refers to the basic element of availability and distribution for the tile table and is comprised of record partitions for the column family *Tile*. Typically, column-based NoSQL databases such as HBase achieve scalable storage of mass data and balanced data distribution via different *Region* divisions, and features such as MemStore and HLog mechanisms can ensure high efficiency for reading and writing tile data and high reliability of data.

When querying and acquiring tile maps with different access patterns, the *RowKey* of the tile map to be accessed first needs to be determined according to the document identifier of the vector map, the level of tile data, and the row and column number. These parameters are passed to the *GMK* function to generate a hash-based *RowKey*. Then, the *RowKeys* of the tile map to be queried are submitted to the master node in the NoSQL cluster. The master node, which maintains the global tile metadata, processes the received information and quickly returns the target data node path in which the tiles are stored. After that, the tile requestor can obtain the desired tile data by directly accessing the data node through *RowKey* matching. This management mechanism can effectively lighten the increased workload via high concurrency and can also improve the efficiency of storage and the access of tile map data.

5.2. Vector Tile Map Parallel Clipping and Query Framework Based on MapReduce

Based on the distributed NoSQL tile data storage model, we propose a framework that is based on MapReduce for parallel clipping and fast scheduling of map tiles. This framework aims to improve the access efficiency of mass tile data. As shown in Figure 9, the bottom of the framework is a NoSQL storage cluster (e.g., an HBase database), among which multiple region hosts as data nodes are used to store distributed tile map data, deploying different tile records in the column-family stores (hereinafter referred to as TCFS, e.g., HTable of HBase) according to our tile data model. Record sets in TCFS are split into multiple independent partitions—tile regions, according to region unit capacity—and are stored in different data nodes to which their respective regions were allocated. The coordination service of NoSQL (such as ZooKeeper) synchronizes tile data regions among different data nodes, maintaining the mapping relationship between global tile datasets and each part in the data nodes, ensuring the logical integrity of the global tile dataset. The master node performs the unified allocation and scheduling of the storage resources of tile data on different data nodes according to regions. The advantage is that accessing multiple map tile data is directly performed by the corresponding distributed data nodes. In addition, the data redundancy replica mechanism supported by the distributed file system (e.g., HDFS) ensures the reliability of tile data. Based on this framework, we realized the parallel generation and retrieval methods of vector map tiles using the model of MapReduce.

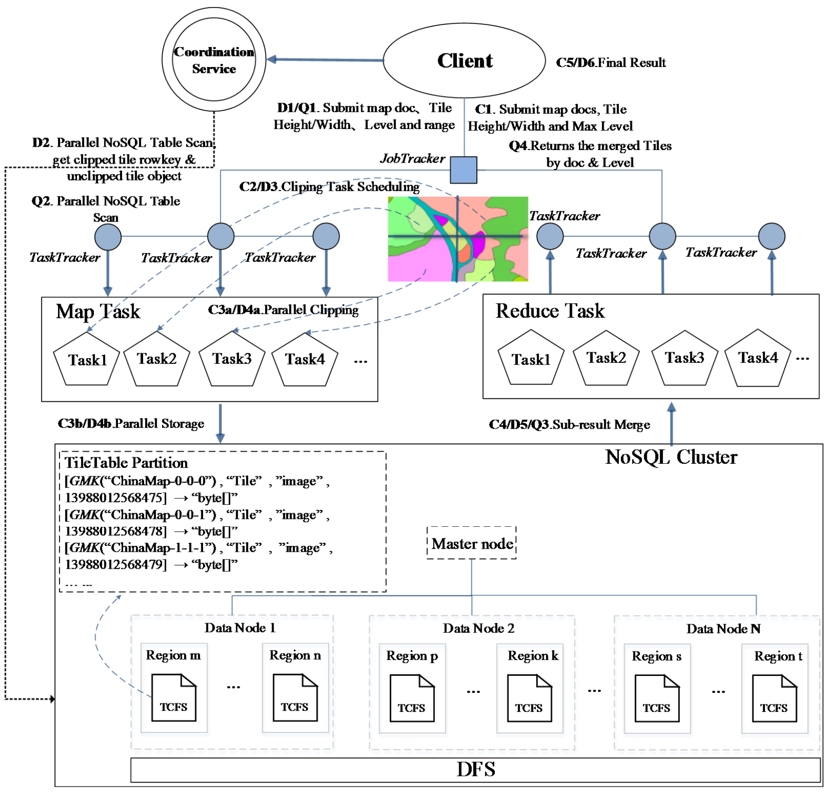
Coordination Service
Client
C5/D6.Final Result
D1/Q1. Submit map doc、Tile Height/Width、Level and range
C1. Submit map docs, Tile Height/Width and Max Level
D2. Parallel NoSQL Table Scan, get clipped tile rowkey & unclipped tile object
JobTracker
Q4.Returns the merged Tiles by doc & Level
C2/D3.Cliping Task Scheduling
Q2. Parallel NoSQL Table Scan
TaskTracker
TaskTracker
TaskTracker
TaskTracker
TaskTracker
TaskTracker
Map Task
C3a/D4a.Parallel Clipping
Task1
Task2
Task3
Task4
...
Reduce Task
Task1
Task2
Task3
Task4
...
C3b/D4b.Parallel Storage
C4/D5/Q3.Sub-result Merge
NoSQL Cluster
TileTable Partition
[GMK("ChinaMap-0-0-0"), "Tile", "image", 13988012568475] → "byte[]"
[GMK("ChinaMap-0-0-1"), "Tile", "image", 13988012568478] → "byte[]"
[GMK("ChinaMap-1-1-1"), "Tile", "image", 13988012568479] → "byte[]"
... ...
Master node
Data Node 1
Region m
TCFS
...
Region n
TCFS
Data Node 2
Region p
TCFS
...
Region k
TCFS
...
Data Node N
Region s
TCFS
...
Region t
TCFS
DFS

Figure 9. MapReduce-based tile map parallel processing framework.

5.3. Tile Map Parallel Clipping Algorithm

The map tile-clip operations generally fall into two categories, static clip and dynamic clip, according to different application scenarios. Static tile clipping mainly refers to pre-generating a tile map before a user accesses the data and distributing the tile map among tile databases or some type of file systems. At present, the main handling method of static clipping is to use either a sequential processing mechanism or a multi-process computation method, and both are based on standalone nodes. However, the extension of computational capabilities is seriously limited by the hardware capacity of a standalone computer, particularly when a vector map consists of many sub-documents or a map document contains a large number of levels; in such cases, it often takes a long time to clip maps, sometimes even hours. This could lead to extreme inefficiencies in static tile generation.

In our framework, by using the MapReduce distributed data processing model, a parallel algorithm is presented to execute tile generating operations (See steps C1 to C5 in Figure 9): a client submits a clip task and uploads the computing resource—map *doc* set, which includes multiple vector map documents and task configuration information (tile size: Tile *height*/*width*, the maximum level of tiles: *maxlevel*, etc.) to the task execution endpoint. *JobTracker*, the task scheduling node of the MapReduce framework, would accept the task and then assign it to various job execution nodes—*TaskTracker*. *TaskTracker* will automatically download appropriate computing resources from the task resources pool for parallel execution of sub-tasks. First, *Mapper* instances are constructed to perform *Map* calculations. As shown in Figure 10, the main process of the *Map* algorithm is described as follows: *t* represents a tile unit in a specified level of the given map document. The *clip* function performs tile generating calculation for each ***t*** to output corresponding tile data which are then put into a TCFS record table (i.e., HBase table). Each map-task's computing result is a Key/Value object and identified by a tile record access number (i.e., *rk*); then, all the *rk* keys are sent to the MapReduce framework for the next *Reduce* execution phase. In the Reduce stage, *TaskTracker* will perform the *Reduce* operation when receiving the execution request of sub-tasks, and parallel merging will be performed on the tile access numbers that represent tile units that were successfully yielded according to different vector map documents. The final result set will be returned to the client after merging. In the entire process, through execution of the two stages, *Map* and *Reduce*, the MapReduce framework accomplishes the parallel computation work for large-volume map tile data.

```
1: class MAPPER
2:     method MAP(doc $d$, maxlevel $m$)
3:         for all level $l \in$ levelset ($m$) do
4:             for all tile $t \in$ tiles($l$, $d$) do
5:                 EMIT(tuple $\langle d, l \rangle$, clip $t$)    ▷ Emit clip task for each tile unit

1: class REDUCER
2:     method REDUCE(tuple $\langle d, l \rangle$, rksets [$rk1$, $rk2$, . . .])
3:         $P \leftarrow$ new LIST
4:         for all rowkey $rk \in$ rksets [$rk1$, $rk2$, . . .] do
5:             $P$.ADD($rk$)
6:         EMIT(doc $d$, LIST $P$)    ▷ Emit tile rowkey list for each map doc
```

Figure 10. Pseudo-code for the static tile-clip algorithm in MapReduce.

As to the clip function implementation details, the *rendering* function is provided by MapGIS K9, one of the most popular GIS platform softwares in China. The MapGIS *rendering* function's prototype statement is listed as follows:

void rendering(Layer<List> layers, SLD<List> descriptors, GeoEnvelope domain, Rectangle png_size,String output_path);

where *layers* represent multiple map documents that can be specified, *descriptors* represent corresponding SLD (Styled Layer Descriptor) files (i.e., each map document has its own SLD file), *domain* represents geographical rectangle for map rendering, *png_size* represents the desired output PNG picture's size and is fixed as the tile unit size 256 × 256 in the clip function, *output_path* represents the output file path for the rendered picture.

Because the *clip* function passes parameters like map documents associated with SLD files, *level*, *xtile* (tile index in x axis) and *ytile* (tile index in y axis), the following Equations (2)–(4) are used to convert *level*, *xtile* and *ytile* into the *geographical rectangle* comprising latitude and longitude ranges. After these transformations, the rendering function is able to be invoked by the clip function and the tile picture could be thus rendered correctly. For example, the geographical rectangle of the tile (*level* = 1, *xtile* = 0, *ytile* = 0) is calculated as ($longitude(2^1, 0)$ $latitude(2^1, 0)$, $longitude(2^1, 1)$ $latitude(2^1, 1)$) = (−180° 85.05°, 0° 0°).

$$n = 2^{level} \tag{2}$$

$$longitude(n,xtile) = xtile/n \times 360.0 - 180.0 \tag{3}$$

$$latitude(n,ytile) = (\arctan(\sinh(\pi \times (1 - 2 \times ytile/n)))) \times 180/\pi \tag{4}$$

In a dynamic tile-clip manner, vector datasets need not be previously processed; instead, the corresponding tile map is dynamically generated according to the map zoom level, query range, etc., only when users access vector map tiles, and then the generated map tiles are stored to the NoSQL TCFS store table. The main difference with static tile production is that tile data are dynamically generated and loaded when users continuously access a series of tiles across multiple map levels or multiple non-adjacent tiles or some specific areas, not using tiles generated in advance. Because the map data do not need pre-processing, this method is more flexible and convenient and saves storage space on the server side. However, the task processing mode requires real-time tile generation, so the background processing algorithm must have low time complexity. Currently, most dynamic tile-clipping implementations are also based on a quadtree index structure; when users randomly access a piece of a tile or multiple tiles of a region, they could first check whether the requested data exists in tile datasets or not; if not, then the tile-clipping operation would be performed on the vector map data using the quadtree-based map clip algorithms. The scenario shown in Figure 11 is a quadtree-based tile map partition graph in a dynamic request session, where the corresponding geographic location of this tile map is Wuhan City of China.

In the NoSQL-based tile processing algorithm framework we have proposed, a parallel dynamic clipping method on the basis of the MapReduce model is also considered; the algorithm steps (D1 to D6) are shown in Figure 10. The basic principle of each computation unit is roughly the same as static tile clipping—the inputs are the tile's *level* and the map's *range* generated with the map *doc* set accessed dynamically. After tasks and related computing resources are submitted to the execution end, *JobTracker* of the MapReduce algorithm framework, it would notify each sub-task execution end, i.e., *TaskTracker*, to fetch their corresponding map *doc* and allocated resources; each *TaskTracker* constructs a *Mapper* object instance to execute the *Map* method, as shown in Figure 12, and the *tiles* function creates tile object collections using the spatial extent m within level $\boldsymbol{l}$ of the map document object d. For each tile object t in *tiles*, obtain the tile's binary from the global tile record table of NoSQL; if its value is empty, then call the *clip* function to generate the object t immediately; the calculation results are stored in the tile data table, and its access number rk is encapsulated as an object that could be processed in *Reduce* stages; if the value is not empty, it indicates that the tile data have already been generated; then, package the record number into Key/Value objects and dispatch them to *Reduce* stages.

The execution mechanism of the *Reduce* course on subtasks is similar to that of static tile processing, and the *TaskTracker*'s end will start the *Reduce* operation execution when qualified objects are received. The successfully clipped tile units and those already existing in the tile database will be parallel merged according to different vector map documents, and the final result set will be sent back to the requestor after reduction.

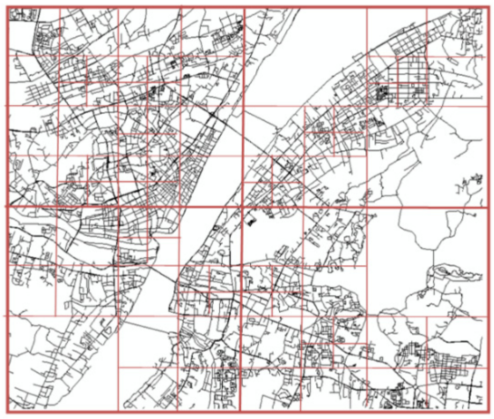

Figure 11. Quadtree-based partition graph of a vector map in dynamic tile clipping.

```
1: class MAPPER
2:   method MAP(doc $d$, level $l$, range $m$)
3:     $G \leftarrow$ GLOBETILETABLE        ▷ Initialize $G$ with Globe tile table
4:     for all tile $t \in$ tiles($l$,$d$,$m$) do
5:       if $G$.GET($t$) = ∅ then
6:         EMIT(tuple $\langle d,l \rangle$, clip $t$)  ▷ Emit clip task for each target tile unit
7:       else
8:         $rk \leftarrow G$.GETROWKEY($t$)
9:         EMIT(tuple $\langle d,l \rangle$, $rk$)     ▷ Emit rowkey for each clipped tile unit

1: class REDUCER
2:   method REDUCE(tuple $tp$, rksets [$rk1$, $rk2$, . . .])
3:     $P \leftarrow$ new LIST
4:     for all rowkey $rk \in$ rksets [$rk1$, $rk2$, . . .] do
5:       $P$.ADD($rk$)
6:     EMIT(tuple $tp$, LIST $P$)        ▷ Emit requested tile rowkey list for each level
```

Figure 12. Pseudo-code for the dynamic tile-clip algorithm in MapReduce.

5.4. Parallel Query of the Tile Map

The parallel query algorithm of the tile map (See Figure 13), primarily implemented by means of the MapReduce computing paradigm, yields the parallel retrieval of tile objects in our NoSQL tile storage model. The input of this algorithm is the query condition collection $Q = \{c \mid c = <d, l, m>\}$, where the triple c is comprised of three query conditions: d, l, and m, which denote the map document object, zoom level and coverage, respectively. Subtask execution ends (i.e., *TaskTracker*) obtain query objects from the collection set Q and then download computing resources and quickly scan the tile table, constructing a tile object subset according to the query object. For each query object ***t*** in the subset, the associative *RowKey* in the global tile data table is obtained and then packaged into computing units handled in the *Reduce* phase according to the query level. The *EMIT* function dispatches these units to the MapReduce task scheduler (i.e., *JobTracker*). It is worth reiterating that if the query result is null, the *EMIT* function would set the value of the emitted object to zero. The algorithm process in the *Reduce* stage is similar to that of dynamic tile clipping: composing a tuple according to the map document and query level, merging query results, and returning the final result to the requestor.

```
1: class MAPPER
2:   method MAP(doc $d$, level $l$, range $m$)
3:     $G \leftarrow$ GLOBETILETABLE              ▷ Initialize $G$ with Globe tile table
4:     for all tile $t \in$ tiles($l$,$d$,$m$) do
5:       if $G$.GET($t$) $\neq \emptyset$ then
6:         $rk \leftarrow G$.GETROWKEY($t$)
7:         EMIT(tuple $\langle d,l \rangle$, $rk$)    ▷ Emit rowkey for each target tile unit
7:       else
8:         EMIT(tuple $\langle d,l \rangle$, 0)     ▷ Emit null for each unclipped tile unit

1: class REDUCER
2:   method REDUCE(tuple $tp$, rksets [$rk1$, $rk2$, ...])
3:     $P \leftarrow$ new LIST
4:     for all rowkey $rk \in$ rksets [$rk1$, $rk2$, ...] do
5:       $P$.ADD($rk$)
6:     EMIT(tuple $tp$, LIST $P$)      ▷ Emit requested tile rowkey list for each level
```

Figure 13. Pseudo-code for the parallel tile query algorithm in MapReduce.

6. Experiments

To evaluate the performance of the model we have proposed in the previous section in a real environment, we set up a test environment using the China Geological Survey Information Grid (hereinafter referred to as the CGS-Grid) and created a verification prototype. Based on this, we carried out a series of experiments. These experiments mainly compared our framework with the traditional model, which adopts a local tile cache mechanism and distributed file system-based management model in terms of tile storage efficiency and performance.

Our testing prototype system is built on the CGS-Grid tile data center, the system architecture that is shown in Figure 14; it is mainly composed of a tile data cluster with 8 NoSQL servers, where data nodes are connected with each other via Gigabit Ethernet. Each computation end is configured with one 2.8-GHz Dual-Core Intel processor, 4 GB of 1066-MHz DDR3 ECC memory and 1 TB of disk storage, running the CentOS Linux 5.8 64-bit operating system. One of these NoSQL nodes is used as a distributed tile storage controller and task scheduler (i.e., *NameNode* and *JobTracker* in Figure 14), whereas seven other nodes are selected for data storage and job execution (i.e., *DataNode* and *TaskTracker* in Figure 14). The tile cluster uses the Hadoop 1.1.2 platform and HBase 0.96 NoSQL database, which provide distributed NoSQL stores. To facilitate the comparison between our method and others, we also prepared one high-performance computer, which offers a standalone tile-clipping service using the classic model, and its configuration is as follows: 24 Intel Dual-Core 2.8-GHz processors, 64 GB memory, and Fiber Channel-based Storage Area Network (FC SAN) with 50 TB of storage capacity. In these experiments, we used the unified public datasets of the China Geological Survey [25], which include 110 vector map documents (covering geophysical, geochemical and many other geological survey topics). The size of the vector datasets is over 300 GB, and the maximum level of LOD model-based tiles reaches 25. Tile map clipping and search functions are wrapped into restful GIS services in accordance with different processing frameworks, including the HPC-based standalone GIS tile service, the Hadoop Distributed File System (HDFS)-based distributed tile processing system, and our tile processing algorithm framework based on MapReduce and NoSQL.

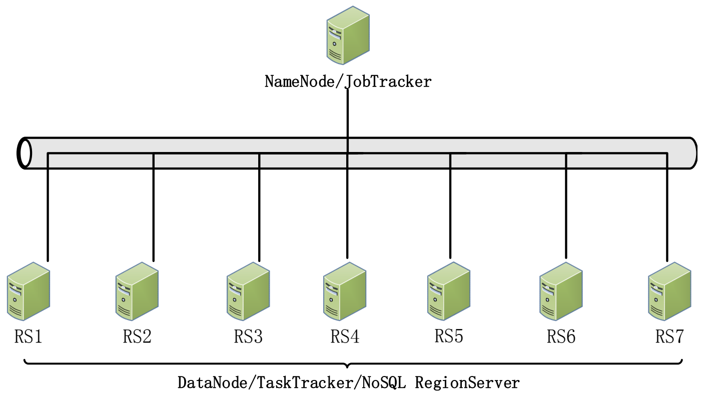

Figure 14. NoSQL cluster-based tile processing architecture.

Using the above-mentioned tile services, a web map client (See Figure 15) is developed to provide geological map services suitable for three different tile data storage and management models. Based on this, we conducted a group of experiments to compare the parallel processing capabilities and scalability between our NoSQL-based method and other common processing patterns. The former two experiments are used to evaluate the parallel tile query processing capabilities of our algorithm framework, whereas the latter two are used to assess its parallel tile-clipping performance.

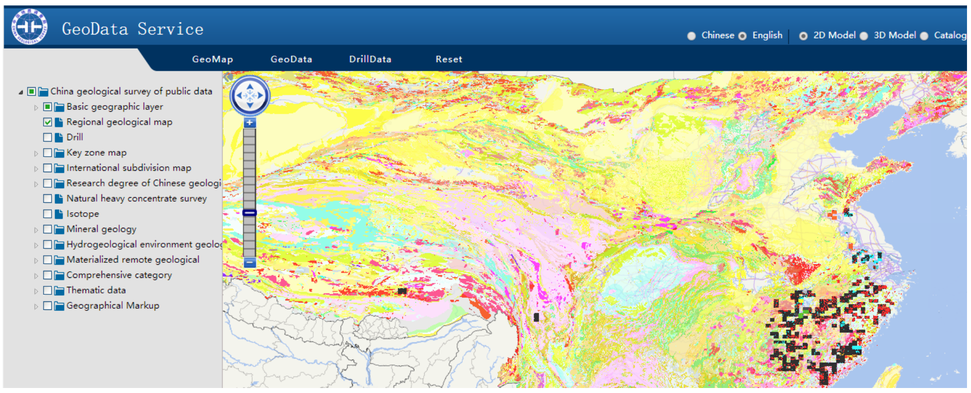

Figure 15. Web portal for the tile map of a China geological survey in the GSI Grid.

6.1. Data Description

Our experiment is based on the real datasets of the China Geological Survey Information Service Platform [25]. The datasets are classified into 39 major categories and contain 247 thematic geological datasets (as shown in Table 1), including multi-scale geological vector maps of the ten most important geological spatial databases of China (covering 1:200,000 and 1:500,000 geological map, 1:200,000 hydrological geological map, 1:200,000 geochemical map, mineral deposits, natural heavy mine, isotope distribution, 1:500,000 environmental geological map and geological exploration databases) and dozens of thematic geological survey datasets. The volume of datasets has reached about 1 TB [26] and is still expanding continuously.

Table 1. Data catalogues of China Geological Survey Information Service Platform.

ID	Layer Type
1	Stream sediment geochemistry exploration dataset (scale: 1:50,000/1:100,000/1:200,000/1:500,000)
2	Soil geochemical exploration dataset (scale: >1:50,000/1:50,000/1:100,000/1:500,000)
3	Petro-geochemical exploration dataset (scale: 1:200,000/1:500,000)
4	Comprehensive geochemical exploration dataset (scale: $\leq$1:100,000/>1:100,000)
5	1:200,000 scale geochemical exploration datasets of China (including 37 elements and compounds: Al_2O_3/Au/B/Ba/Be/Bi/Cao/Co/Cr/Cu/F/Fe_2O_3/K_2O/Li/MgO/Mn/Mo/Na_2O/Nb/Ni/P/Sb/SiO_2/Sn/Sr/Th/Ti/V/W/Zn/Zr/Ag)
6	Geomagnetic survey dataset (scale: <1:25,000/$\geq$1:25,000)
7	Geo-electric survey dataset (scale: $\leq$1:100,000/1:25,000/1:100,000/$\geq$1:25,000)
8	Ground radiation survey dataset (scale: $\leq$1:100,000/>1:100,000)
9	Seismic survey dataset (scale: $\leq$1:100,000/>1:100,000)
10	Airborne magnetic survey dataset (scale: 1:1,000,000/1:500,000/1:200,000/1:100,000/>1:100,000)
11	Airborne electromagnetic survey dataset
12	Airborne radioactive survey dataset
13	Other methods of geophysical survey dataset
14	Deep geophysical investigation dataset
15	Gravimetric dataset (scale: 1:1,000,000/1:500,000/1:200,000/1:100,000/>1:100,000)
16	Comprehensive geophysical exploration dataset (scale: $\leq$1:100,000/1:25,000/1:100,000/$\geq$1:25,000)
17	Engineering geological investigation dataset (scale: $\leq$1:100,000/>1:100,000)
18	Urban environmental geological investigation dataset
19	China geological hazard dataset (1:2,500,000 scale) (abrupt severe geological hazards/disaster-affected level datasets for landslide, rockfall and debris flow hazards/synthetic evaluation dataset for abrupt geological hazards)
20	Geological survey dataset of mining environment
21	Ecological environment geological survey dataset
22	Water pollution geological survey dataset
23	Nonmetallic mineral exploration dataset
24	Precious metals mineral exploration dataset
25	Ferrous metals mineral exploration dataset
26	Coal mineral exploration dataset
27	Nonferrous metals mineral exploration dataset
28	Comprehensive mineral exploration dataset
29	Regional geological survey datasets (scale: 1:200,000/1:250,000/1:500,000/1:1,000,000), mining area geological mapping dataset cum comprehensive regional geological survey dataset
30	Hydro-geological survey dataset (scale: 1:200,000/1:250,000/1:500,000/1:1,000,000) and other types hydro-geological dataset (scale: >1:100,000/$\leq$1:100,000)
31	Hydro-geological, engineering and environmental geological survey dataset and comprehensive geochemical survey dataset (scale: 1:50,000/1:100,000)
32	Natural heavy minerals dataset
33	China isotopic dating dataset
34	Drill dataset
35	China important thermal springs dataset (1:6,000,000 scale)
36	China important karst cave dataset (1:4,000,000 scale)
37	China geological park dataset (1:4,000,000 scale)
38	Qinghai-Tibet Plateau geological dataset (1:250,000 scale)
39	China groundwater resources dataset

6.2. Experiment 1—Concurrency

The first experiment compared the tile data access performance of the standalone file cache-based method, the distributed file system (Hadoop HDFS)-based method and our proposed NoSQL-based management framework for tile map dataset processing.

In these experiments, we primarily utilized the same tile map sets (the China geochemical map set, at a scale of 1:200,000, contains 32 map documents; the vector data size of the set is approximately 20 GB, and the tile files generated by the standalone clipping operation, with a size of 46.3 GB, are used in one standalone HPC, a Hadoop cluster and the NoSQL cluster framework mentioned above) and conducted performance tests on three access patterns; in addition, the HDFS replica switch was closed during the test. The first pattern is accessing one particular map tile based on the identification triple of the tile object, and the corresponding experiment sets the single tile unit at row 0, column 0 of level 0 as the target area; the second mode accesses multiple tile maps of the same level by using the rectangle range condition, and the access scope in the experiment covers the four total tiles of map level 1; the third method is accessing multiple tile maps crossing several levels according to the rectangle range, and in the experiment, we used 21 total tiles covering level 0 to level 2 as the query target. The average access times of multiple vector map tile documents in the three architectures under different users and access models are analyzed. The experimental results are shown in Table 2.

Table 2. Average response time of three concurrent access patterns (seconds).

Virtual Users	**Single-Tile Query**			**Multiple-Tile Query in the Same Level**			**Multiple-Tile Query across Multiple Levels**		
	File System	**HDFS**	**NoSQL**	**File System**	**HDFS**	**NoSQL**	**File System**	**HDFS**	**NoSQL**
10	0.013	0.019	0.016	0.033	0.045	0.036	1.125	1.252	1.147
50	0.135	0.180	0.143	0.362	0.506	0.425	1.973	2.094	1.982
100	0.304	0.297	0.263	0.731	0.636	0.520	3.429	3.303	3.075
200	0.765	0.629	0.554	1.049	0.883	0.725	5.641	4.786	4.142
300	0.942	0.986	0.873	1.275	1.143	1.012	6.272	5.681	5.093
400	1.252	1.026	0.943	1.536	1.464	1.258	6.941	6.548	5.586
500	1.460	1.245	1.123	1.840	1.644	1.460	7.267	6.863	6.092

From Figures 16–18, we can see that when a concurrent query is on a small scale, the standalone system is considerably more efficient than distributed architecture. In the case that the number of concurrent requests is less than 50—either concurrent queries of a single-tile unit or those of multiple tiles in the same level—the average response times of the distributed architecture are generally lower than those of the standalone server architecture. Nevertheless, the efficiency of our NoSQL-based framework is higher than that of the distributed file system architecture. Obviously, in this scenario, the underlying network communication cost of the distributed architecture is a higher proportion of the total execution time, and the time cost of data block scheduling across multiple nodes of the distributed file system is higher than that of parallel tile retrieval using the MapReduce model in our tile processing framework. However, when concurrent sessions reach more than 100, the average access time of tile data in the distributed environment is less than that of the local cache mode; this indicates that the concurrent scale has a specified threshold value and that when the threshold is reached, a gradual increase in virtual users, the complexity of tile querying (e.g., from simple queries in the same level to queries covering multiple levels) or the total number of tiles to be queried will cause the efficiency of the mass query in the distributed environment to be much higher than that of the local file cache mechanism. In addition, under different modes of concurrent queries, the NoSQL-based parallel framework has better performance than the distributed file system-based framework. All of the above demonstrate that our NoSQL-based parallel processing model is suitable for fast data processing in high-concurrency and large-volume map tile scenarios.

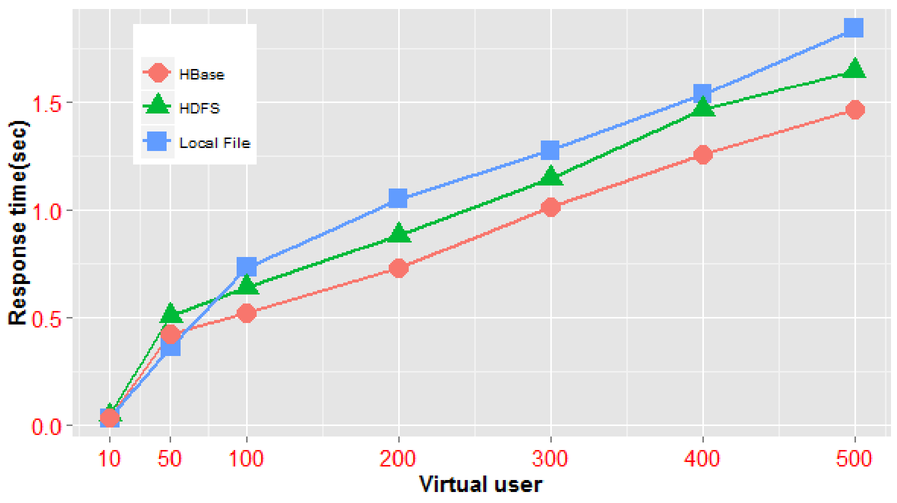

Figure 16. Response time of a single-tile query by concurrent service requests.

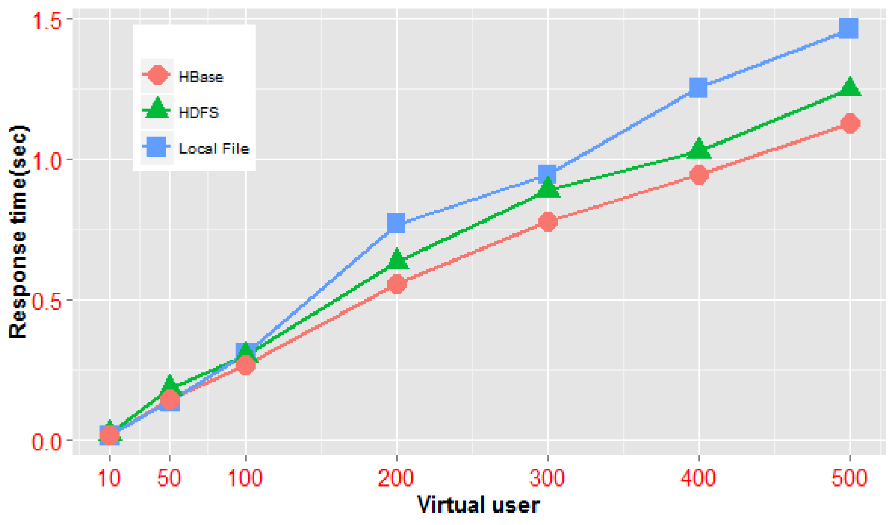

Figure 17. Response time of a rectangle-range query in the same level.

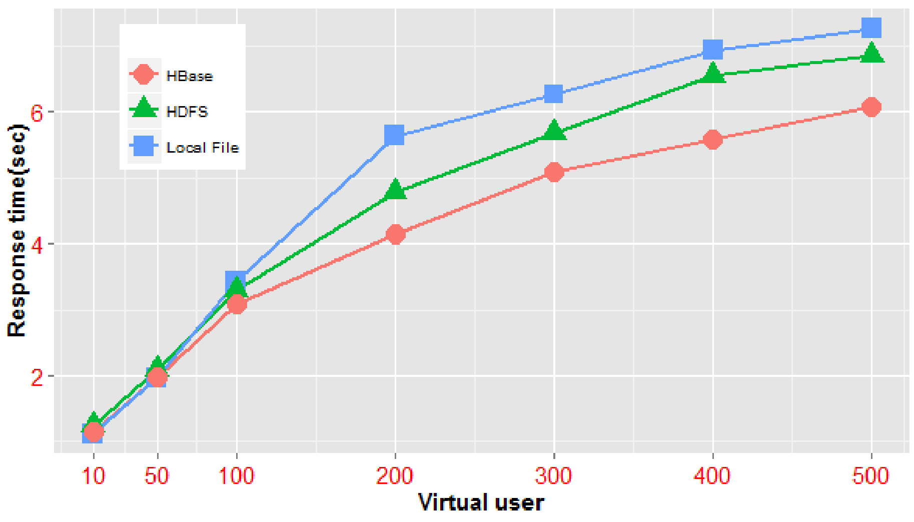

Figure 18. Response time of a cross-level rectangle-range query.

6.3. Experiment 2—Scalability

In addition to the number of concurrent sessions and the ways in which tile data are accessed, the processing capabilities of tile map querying are also dependent on the number of participating nodes in the parallel processing, especially in large-scale vector tile map application scenarios. Experiment 2 is mainly used to explore the relationship of the response time of query requests and the number of processing nodes of the NoSQL architecture under the same query conditions. In the experiment, we used a China geological vector map on a scale of 1:200,000 (the size of the vector data is 7 GB, 17 tile levels are generated using the NoSQL framework, and the capacity of the tile record table of NoSQL reaches approximately 17.3 GB). We gradually increase the number of cluster nodes, one at a time, from three nodes at the beginning to seven nodes. Every time a node is added, a data balancer will balance the distribution of the global tile data in the NoSQL table over all *Region* hosts, which can yield better load balance and improve storage scalability. The generation rules of query objects are described as follows: for each query, one tile unit of each level from level 3 to level 17 is randomly selected to compose a cross-level query objects set (which includes 15 tiles). We conducted the experiment under different concurrent access sessions with the query objects and recorded the average response time of the framework. The results are shown in Table 3.

Table 3. Response time of a complex query with multiple nodes by concurrent requests.

Nodes / Users	3	4	5	6	7
10	0.948	0.949	0.950	0.949	0.953
50	1.283	1.280	1.287	1.285	1.283
100	1.562	1.560	1.556	1.548	1.522
200	1.982	1.983	1.962	1.931	1.904
300	2.586	2.543	2.492	2.456	2.421
400	3.663	3.526	3.364	3.181	3.042
500	4.247	4.132	3.916	3.785	3.472

The experimental data in Table 3 can be expressed as a line chart (Figure 19); as we can see, when the number of concurrent sessions is less than 100, the average response time of a query is not significantly affected by the increase of the node members. However, when the number of virtual users reaches 100, the more nodes we have participating in the processing, the faster our framework responds to the query. In the scenarios of small-scale concurrent query tasks, the effect of improving the processing efficiency by increasing the number of NoSQL nodes is not apparent; the reason for this is that the parallel acceleration gained by increasing the processing nodes is partly offset by the concurrently increased cost of network communication. When the scale of the concurrent query grows to a particular threshold (e.g., 200 users in Figure 19), the communication overhead directly increased by new nodes becomes negligible compared to the parallel processing acceleration capabilities brought by the expansion. The experimental results have shown that by increasing the processing nodes properly, we can distribute query tasks to minimize the load on any single processing node and thus increase the supportable number of parallel tile query tasks. All of this can effectively expand the parallel processing capabilities of our NoSQL computing framework.

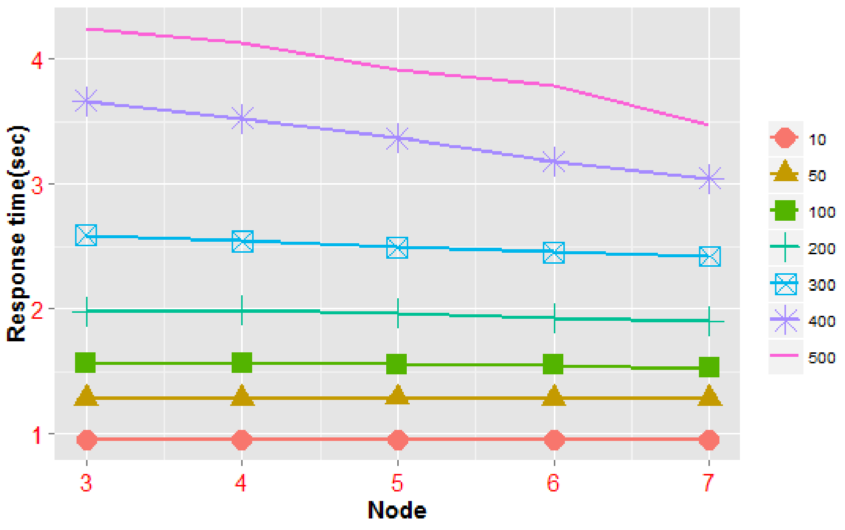

Figure 19. Response time of concurrent tile requests by different quantities of processing nodes.

Let u denote the number of concurrent sessions initiated by a tile query transaction, and let S be the number of tiles in the query tasks. S can be defined as follows:

$$S = \sum_{i=1}^{u} n_i \tag{5}$$

where n_i represents the number of tiles requested by the ith concurrent session. For our NoSQL-based framework with N processing nodes (where each node has o processor cores), the average response time of the tile query operation T_r is given generally by:

$$T_r = S \times \frac{1}{N^2} \times \frac{1}{o} \times \left(\frac{t + c_m}{\mathrm{m}} + \frac{p + c_r}{r} \right) \tag{6}$$

where t is the average query time of a single tile object in the NoSQL table using our data storage model (in our model, a tile record's *RowKey* is designed to be generated according to the row and column number of each tile unit, thus avoiding a full-table scan when performing a query operation), and p denotes the average time of the merge operation on query results, whereas m and r represent the number of *Map* and *Reduce* jobs, respectively, on one single node. In addition, c_m and c_r are random variables that obey exponential distributions with parameters λ_m and λ_r, respectively. They refer to the time overhead caused by the uncertainty factors, such as communication latency, disk addressing and busy clients when performing each *Map* or *Reduce* operation.

Obviously, the execution time T_r is mainly proportional to the concurrent users, the total number of tiles, the execution time of the query program the and network communication time, and it has an inverse relationship with the number of nodes involved in the calculation. When the scale of S is small, with the increase of N, the reduction scale of T_r is not significant, but when the S scale reaches a certain level, such as the continuous growth of N, the effect on the reduction scale of T_r is more obvious. Consequently, our distributed tile management framework is suitable for application scenarios with high concurrency and a large amount of data processing. In these scenarios, increasing the number of parallel processing nodes can effectively improve the efficiency of massive tile queries. Moreover, to make the tile scheduling as efficient as possible, we need to adopt the method of improving the hardware configuration, such as increasing the bandwidth of the network adaptor and the core number of processors, etc.

6.4. Experiment 3—Static-Clipping Performance

Experiment 3 compared the computing performance of three different static tile-clipping architectures—a traditional high-performance server, distributed file-based processing system and our NoSQL-based parallel framework. In the experiment, we selected the same vector datasets as in Experiment 2, generating 0 to 6 tile levels for a total of 981 tile map units through the static clipping operation requested by the client.

Based on closing the HDFS replica function switch, we compared the average execution time of the static clipping of the single server mode and distributed computing mode, and the results shown in Figure 20 indicate that the local file cache method spent the most time on processing (264.3 s), whereas the MapReduce diagram-based NoSQL framework has obvious advantages (92.7 s) in efficiency. We can clearly see that the average transaction completion time of performing the MapReduce clipping task based on the HDFS framework is slightly faster than that of the centralized model, but the advantages are not obvious; the reason for this is that the cross-node data access costs brought by the increased tile files in the processing should not be neglected. However, our NoSQL framework with MapReduce parallel algorithms of static tile clipping on the vector dataset is more efficient.

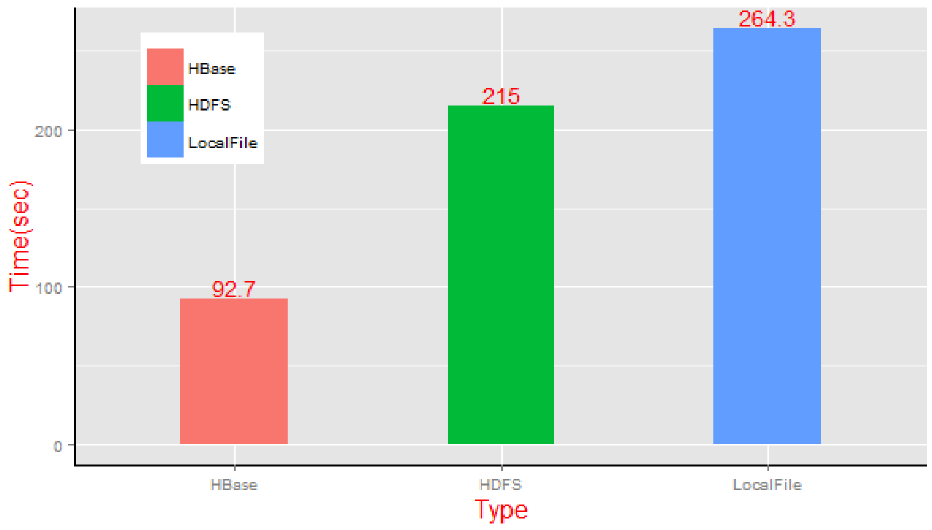

Figure 20. Average execution time of the static-clip operation by three processing models.

6.5. Experiment 4—Dynamic-Clipping Performance

Experiment 4 compared the task completion time of a dynamic-clipping transaction by the traditional method of a high-performance server, the Hadoop distributed file system processing method and our proposed approach, i.e., dynamic tile generation using a NoSQL cluster. By using the same real datasets as Experiments 2 and 3, we make dynamic-clipping requests for China geological vector map data in the three architecture environments; according to the one tile unit per level rule, we made requests to ten tiles randomly selected from levels 3 to 12 of the vector map sets and then analyzed their dynamic clipping task completion time as shown in Figure 21.

The execution of the aforementioned three methods is more time-consuming in our experiments for their dynamic clipping characteristics than their corresponding static methods. In the case of small-scale concurrent requests, the difference of average time consumed for dynamic clipping between the single processing architecture and the HDFS distributed architecture was smaller when generating 10 random tiles; the execution times were 13.47 s and 12.68 s, respectively. Furthermore, besides the operation cost of tile clipping, the store operation for real-time generated tile data also brought a certain amount of time overhead, particularly in the case of enabling the replica function of the HDFS file system (the time expenditure of the HDFS method was even greater than that of the standalone mechanism). However, the advantages of the NoSQL architecture are obvious, as the time expenditure of parallel dynamic clipping was just 8.73 s in the experiment.

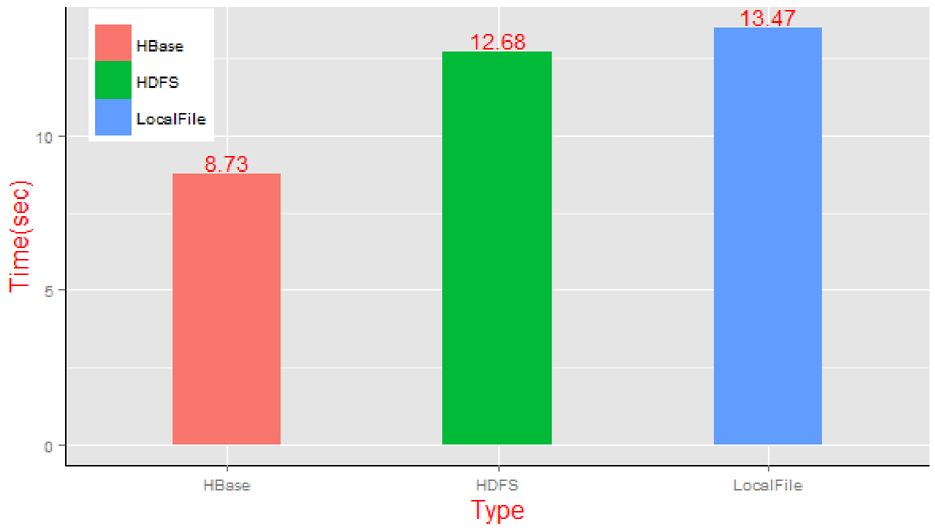

Figure 21. Average response time of dynamic tile clipping by three methods.

7. Conclusions and Future Work

This article has introduced a parallel processing method for vector tile data that can be applied to clipping and storage operations for massive data tiles in high-concurrency map service scenarios. A column-family-based distributed tile data storage model was developed for the cases of the construction, manipulation, retrieval, and sharding of large-scale tile data in a NoSQL cluster. This data model addresses the critical need to support high storage efficiency, rapid scaling ability and balanced data distribution.

To improve the processing efficiency of a tiled map in the NoSQL data model, we designed an algorithmic framework for performing distributed clipping and query operations on massive vector tiles. The framework supports both static and dynamic tile clipping in parallel mode for our NoSQL storage model and also implements the fast retrieval of a large-scale tile dataset deployed in NoSQL using the MapReduce programming paradigm. Toward this end, we also embed the parallel processing framework within a high-performance web service environment, which is established on the foundation of the China Geological Survey Information Grid, to develop a tiled map services prototype that can provide clipping and map services for big vector data. Timing experiments reveal that our framework performs well in network scenarios in which vector tiles for geological survey information are distributed over the NoSQL cluster, especially when the growth of both the spatial data volume and the number of concurrent sessions presents challenges regarding the system performance and execution time of tile data processing.

Our NoSQL-based processing method for vector tile data is an initial attempt at developing an effective means to tackle massive volume spatial storage and computing problems in the Cloud. Distributed storage, access and a fast processing model for large-volume three-dimension tile data, especially the exploration of enhancing the efficiency and effectiveness of parallel clipping, geospatial query and analysis, will be one of our next research emphases. In addition, we hope to collaborate with other researchers to explore index structures for high-dimensional spatial data in the NoSQL framework, a complex data querying algorithm for massive spatio-temporal data using the MapReduce programming model, and a coupling mechanism according to the communication and logical relationship between the Map task and Reduce task in favor of the optimization of spatial processing algorithms that contain intensive data operations, such as spatial joins, in future work.

Acknowledgments: This research was supported by the grants from the National Natural Science Foundation of China (41401449, 41501162, 41625003), Scientific Research Key Program of Beijing Municipal Commission of Education (KM201611417004), New Starting Point Program of Beijing Union University (ZK10201501), State Key Laboratory of Resources and Environmental Information System, and Key Laboratory of Satellite Surveying Technology and Application, Chinese National Administration of Surveying, Mapping and Geoinformation.

Author Contributions: Zhou Huang conceived and designed the experiments; Lin Wan performed the experiments; Xia Peng analyzed the data; Zhou Huang, Lin Wan and Xia Peng wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cruz, S.A.B.; Monteiro, A.M.V.; Santos, R. Automated geospatial Web Services composition based on geodata quality requirements. *Comput. Geosci.* **2012**, *47*, 60–74. [CrossRef]
2. Li, H.; Zhu, Q.; Yang, X.; Xu, L. Geo-information processing service composition for concurrent tasks: A QoS-aware game theory approach. *Comput. Geosci.* **2012**, *47*, 46–59. [CrossRef]
3. Zhang, T.; Tsou, M. Developing a grid-enabled spatial Web portal for Internet GIServices and geospatial cyberinfrastructure. *Int. J. Geogr. Inf. Sci.* **2009**, *23*, 605–630. [CrossRef]
4. Zhao, P.; Foerster, T.; Yue, P. The geoprocessing web. *Comput. Geosci.* **2012**, *47*, 3–12. [CrossRef]
5. Evans, M.R.; Oliver, D.; Yang, K.; Shekhar, S. Enabling spatial big data via CyberGIS: Challenges and opportunities. In *CyberGIS: Fostering a New Wave of Geospatial Innovation and Discovery*; Springer: Heidelberg, Germany, 2013; pp. 1–22.
6. Wang, S.; Anselin, L.; Bhaduri, B.; Crosby, C.; Goodchild, M.F.; Liu, Y.; Nyerges, T.L. CyberGIS software: A synthetic review and integration roadmap. *Int. J. Geogr. Inf. Sci.* **2013**, *27*, 2122–2145. [CrossRef]
7. Chen, A.; Di, L.; Wei, Y.; Bai, Y.; Liu, Y. Use of grid computing for modeling virtual geospatial products. *Int. J. Geogr. Inf. Sci.* **2009**, *23*, 581–604. [CrossRef]
8. Tang, W. Parallel construction of large circular cartograms using graphics processing units. *Int. J. Geogr. Inf. Sci.* **2013**, *27*, 2182–2206. [CrossRef]
9. Zhang, J.; You, S. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *Int. J. Geogr. Inf. Sci.* **2013**, *27*, 2207–2226. [CrossRef]
10. Liu, Y.; Padmanabhan, A.; Wang, S. CyberGIS Gateway for enabling data-rich geospatial research and education. *Concurr. Comput. Pract. Exp.* **2015**, *27*, 395–407. [CrossRef]
11. Wang, S. CyberGIS: Blueprint for integrated and scalable geospatial software ecosystems. *Int. J. Geogr. Inf. Sci.* **2013**, *27*, 2119–2121. [CrossRef]
12. Yang, C.; Goodchild, M.F.; Huang, Q.; Nebert, D.; Raskin, R.; Xue, Y.; Bambacus, M.; Faye, D. Spatial cloud computing: How can the geospatial sciences use and help shape cloud computing? *Int. J. Digit. Earth* **2011**, *4*, 305–329. [CrossRef]
13. Yang, C.; Xu, Y.; Nebert, D. Redefining the possibility of digital Earth and geosciences with spatial cloud computing. *Int. J. Digit. Earth* **2013**, *6*, 297–312. [CrossRef]
14. Yue, P.; Zhou, H.; Gong, J.; Hu, L. Geoprocessing in cloud computing platforms—A comparative analysis. *Int. J. Digit. Earth* **2012**, *6*, 404–425. [CrossRef]
15. Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
16. Aji, A.; Wang, F.; Vo, H.; Lee, R.; Liu, Q.; Zhang, X.; Saltz, J. Hadoop GIS: A high performance spatial data warehousing system over mapreduce. *Proc. VLDB Endow.* **2013**, *6*, 1009–1020. [CrossRef]
17. Eldawy, A.; Mokbel, M.F. A demonstration of SpatialHadoop: An efficient mapreduce framework for spatial data. *Proc. VLDB Endow.* **2013**, *6*, 1230–1233. [CrossRef]
18. Guo, D.; Wu, K.; Li, J.; Wang, Y. Spatial scene similarity assessment on Hadoop. In Proceedings of the ACM SIGSPATIAL International Workshop on High Performance and Distributed Geographic Information Systems, San Jose, CA, USA, 2–5 November 2010.
19. Meijer, E.; Bierman, G. A co-relational model of data for large shared data banks. *Commun. ACM* **2011**, *54*, 49–58. [CrossRef]
20. Li, Y.; Kim, G.; Wen, L.; Bae, H. MHB-Tree: A distributed spatial index method for document based NoSQL database system. In *Lecture Notes in Electrical Engineering*; Springer: Dordrecht, The Netherlands, 2012; pp. 489–497.

21. Resch, B. NoSQL suitability for SWE-enabled sensing architectures. In Proceedings of the First ACM SIGSPATIAL Workshop on Sensor Web Enablement, Redondo Beach, CA, USA, 7–9 November 2012.
22. Steiniger, S.; Hunter, A.S. Free and open source GIS software for building a spatial data infrastructure. In *Lecture Notes in Geoinformation and Cartography*; Bocher, E., Neteler, M., Eds.; Springer: Berlin/Heidelberg, Germany, 2012; pp. 247–261.
23. Li, Z.; Hu, F.; Schnase, J.L.; Duffy, D.Q.; Lee, T.; Bowen, M.K.; Yang, C. A spatiotemporal indexing approach for efficient processing of big array-based climate data with MapReduce. *Int. J. Geogr. Inf. Sci.* **2016**. [CrossRef]
24. Huang, Z.; Fang, Y.; Chen, B.; Wu, L.; Pan, M. Building the distributed geographic SQL workflow in the grid environment. *Int. J. Geogr. Inf. Sci.* **2011**, *25*, 1117–1145. [CrossRef]
25. China Geological Survey (CGS). China Geological Survey Information Service Platform. Available online: http://www.gsigrid.cgs.gov.cn/newgsigrid/CGSGeoData/Default.aspx (accessed on 6 September 2016).
26. Li, C.; Song, M.; Lv, X.; Luo, X.; Li, J. The spatial data sharing mechanisms of geological survey information grid in P2P mixed network systems network architecture model. In Proceedings of the Ninth International Conference on Grid and Cloud Computing, Nanjing, China, 1–5 November 2010.