Strategies for Improving Latency and Throughput of the Apache Hadoop Ecosystem for Medical Imaging Data

Shunxing Bao, Andrew Plassard, Bennett Landman and Aniruddha Gokhale
Dept of Electrical Engineering and Computer Science
Vanderbilt University, Nashville,TN 37235, USA
{shunxing.bao,andrew.j.plassard,bennett.landman,a.gokhale}@vanderbilt.edu

ABSTRACT

Traditional medical imaging studies use hierarchical data structures (e.g., NFS file stores) or databases (e.g., COINS, XNAT) for storage and retrieval. The resulting performance from these approaches is, however, impeded by standard network switches, since they can saturate network bandwidth during transfer from storage to processing nodes for even moderate-sized studies. The ecosystem of Apache Hadoop, which is a flexible framework providing distributed, scalable, fault tolerant storage and parallel computational modules, and HBase, which is a NoSQL database built atop Hadoop's distributed file system, is a promising alternative to host and process medical imaging data. Despite this promise, HBase's load distribution strategy of region split and merge is detrimental to the hierarchical organization of imaging data (e.g., project, subject, session, scan, slice).

This paper makes two contributions to address these concerns by enhancing the Apache Hadoop ecosystem for medical imaging applications. First, we propose a new row-key design for HBase driven by the hierarchical organization of imaging data. Second, we propose a novel data allocation policy within HBase to strongly enforce colocation of hierarchically related imaging data. The proposed enhancements accelerate data processing by minimizing network usage and localizing processing to machines where the data already exist. Moreover, our approach is amenable to the traditional scan, subject, and project-level analysis procedures, and is compatible with standard command line/scriptable image processing software. Experimental results for an illustrative sample of imaging data reveals that our new HBase policy results in a three-fold time improvement in conversion of classic DICOM to NiFTI file formats when compared with the default HBase region split policy, and nearly a nine-fold improvement over a commonly available network file system (NFS) approach even for relatively small filesets. Moreover, file access latency is lower than network attached storage.

Middleware '16 Dec 12–16, 2016, Trento, Italy
© 2016 ACM. ISBN 978-1-4503-2138-9…$15.00
DOI: 10.1145/1235

CCS Concepts

•**Theory of computation** → **MapReduce algorithms;** •**Applied computing** → **Imaging;** •**Networks** → *Cloud computing;*

Keywords

Hadoop, HBase, Medical imaging, Grid computing.

1. INTRODUCTION

Traditional grid computing approaches separate data storage from computation. To analyze data, each dataset must be copied from a storage archive, submitted to an execution node, processed, synthesized to a result, and results returned to a storage archive. This is the workflow traditionally adopted in processing medical imaging datasets. However, when imaging datasets become massive, the bottleneck associated with copying and ensuring consistency overwhelms the benefits of increasing the number of computational nodes.

Additionally, although magnetic resonance imaging (MRI) and computed tomography (CT) have become integral to modern medical practice, grand challenges remain in using medical imaging data to their full potential. While vast MRI and CT databases are accumulating in radiology archives (at the rate of nearly 100 million examinations per year in the U.S.), we lack the image processing, statistical, and informatics tools for large-scale analysis and integration with other clinical information (e.g., genetics and medical histories). An efficient mechanism for query, retrieval, and analysis of all patient data (including imaging) would enable clinicians, statisticians, image scientists, and engineers to better design, optimize, and translate systems for personalized care into practice.

In this context, "Big data" can simply be defined as the scale for which traditional database approaches fail, i.e., the performance gains no longer scale with the number of computational nodes, but are limited by the network. Consider, for example, the activity of converting Digital Imaging and Communications in Medicine (DICOM) files to NiFTI (a research file format); if converting a 50 MB volume takes 15 seconds, an ideal Gigabit network ($\approx 100MB/s$) saturates with slightly less than 30 simultaneous processes.

The infrastructure developed to support social networks and e-commerce provides a solution to this problem, which turns out to be simple and relatively inexpensive: one should combine the storage and execution nodes such that each task can be done with minimal copying of data. For example, the

Apache Hadoop ecosystem [1], which provides the Big Data processing capabilities, has been extensively used in these contexts.

Although big data architectures have been applied in online commerce, social media, video streaming, high-energy physics, and proprietary corporate applications, these technologies have not been widely integrated with medical imaging data formats (e.g., DICOM) for medical image processing. Several approaches have followed the path of general machine learning literature and seek to implement algorithms specifically designed to take advantage of big data architecture [10, 5, 27], exploit the MapReduce framework to sift through datasets [23], or use distributed file systems [26, 31]. While such approaches have been effective for genetics studies [31, 8], they have not yet proven effective within current medical image computing workflows.

The fundamental reason for this shortcoming is that substantial resources have been invested in creating existing algorithms, software tools, and pipelines, and hence there is a substantive (often prohibitive) cost associated with algorithm re-implementation and re-design specifically for big data medical imaging. Thus, there is a need for new approaches that will not require algorithm re-implementation while exploit the potential of frameworks, such as Apache Hadoop, that have shown promise in other application domains.

To address these problems, we present a new data model for use with distributed storage and computation systems that provides practical access to distributed imaging archives, integrates with existing data workflows, and effectively functions with commodity hardware. Our approach makes specific improvements to the Apache Hadoop ecosystem, notably HBase, which is a NoSQL database built atop Hadoop's distributed file system. Specifically, we make the following contributions in this paper:

- **New row-key design for Apache HBase:** A hierarchical key structure is proposed to accommodate nested layers of priority for data-collocation.
- **New RegionSplit policy:** A computationally efficient approach is proposed to optimally manage data collocation in the context of the hierarchical key structure.
- **Experimental results:** The proposed innovations are evaluated in the context of a routine image analysis task (file format conversion) on a typical Gigabit research network with 12 nodes.

The performance of this new system is evaluated on small (7 GB) to moderate-sized (530 GB) test cases to characterize the overhead associated with this model and demonstrate tangible gains on widely available network and computational hardware. We believe that the proposed improvements to the Apache Hadoop ecosystem will greatly reduce the technical barriers to performing high-throughput image processing necessary to integrate imaging data into actionable metrics for personalized medicine.

The rest of the paper is organized as follows: Section 2 compares our work with related work; Section 3 provides background on the Apache ecosystem and describes our contributions; Section 4 describes our evaluation approach and presents experimental results; and finally Section 5 presents concluding remarks alluding to ongoing and future work.

2. RELATED WORK

Recent trends indicate a substantial interest in adopting the MapReduce paradigm – and thereby the Apache Hadoop ecosystem – for medical image data processing. Several medical image processing studies have encountered one or more of the trio of computation, storage and network bandwidth bottlenecks, and have developed optimizations to overcome these encountered problems. In this section we focus predominantly on comparing our work with these prior works. Additionally, we also review prior studies that are not necessarily focused on medical imaging. We scope out our comparisons to only those prior works that have leveraged the Apache Hadoop ecosystem.

2.1 Related Work involving Medical Imaging Applications

A recent study [23] illustrates how transitioning the medical image processing computations to the MapReduce paradigm and the Apache Hadoop framework pays rich dividends over traditional processing approaches, which often are sequential in nature. The specific use cases for which the results are reported include (a) the use of support vector machines for optimizing the parameters for lung texture segmentation, (b) content-based medical image indexing, and (c) a 3-D directional wavelet analysis for solid texture classification. Our work differs from this prior work in that not only is our use case different – we focus on mapping DICOM images to NiFTI formats – but more importantly we demonstrate new optimization strategies for the Apache Hadoop ecosystem instead of simply leveraging the default strategies provided by Apache Hadoop, which is the case with most prior efforts. In fact the authors in this related work point out the need to identify opportunities for optimizations, which is precisely the intent of our presented research.

A recent prior work [19] has used the Apache Hadoop ecosystem for content-based image retrieval where the MapReduce paradigm is used to extract vector features of the images. Similar to [23], the authors in this study demonstrate how the Apache Hadoop ecosystem can be used in medical imaging but do not report on any optimizations.

The work reported in [25] is synergistic to our work in that it focuses on the row- versus column-oriented storage issues for DICOM images. The authors highlight the pros and cons of row- versus column-oriented storage policies, and indicate how the complex structure of the DICOM images requires a hybrid mechanism for storage. Specifically, their approach stores frequently used attributes of a DICOM file into row-based layer/store, and optional/private attributes into a column-based store so that it will reduce null values. The motivation stems from the fact that if all DICOM attributes are stored into a row-based store, then a search or joining operation will unnecessarily involve numerous null values thereby adversely impacting efficiency.

The SYSEO project [6] also describes a hybrid row-column data store for DICOM images using similar criteria as in [25] to decide between row- versus column-based storage. Their work was motivated from the need to find alternatives to existing but prohibitively expensive solutions for medical image storage. Moreover, image annotation and query retrieval were additional dimensions that needed improvements in performance.

For our work, we do not treat DICOM file attributes in as much depth as in [25], i.e., we do not need to know the details of the attributes stored in a DICOM file when we store it to HBase; rather we simply store the entire DICOM file to HBase. For our DICOM to NiFTI processing, the processing operation can directly fetch the related attributes from DICOM files and convert them into NiFTI files. It is possible that for other forms of medical imaging applications and data processing, such as image annotations, we may need to incorporate these hybrid storage mechanisms along with our optimizations. However, the current paper does not report on such combined optimizations, which forms a dimension of our future research.

2.2 Related Work in Other Application Domains

Several prior research efforts have proposed different performance optimizations to different elements of the Apache Hadoop ecosystem for application domains beyond just medical image processing. The MHBase project [21] describes a distributed real-time query processing mechanism for meteorological data with the intent to provide safe storage and efficient.

The data in Internet of Things are always large volume, which update frequently and are inherently multi-dimensional. The work in [22] proposes an optimization based on high update throughput and query efficient index framework (UQE-Index) including pre-splitting the HBase region for reducing the cost of data movement. The work in [20] addresses the problem of the HBase multidimensional (upto four-dimension) data queries in Internet of things with better response time.

A recent work [30] demonstrates an optimized key-value pair schema for speeding up locating data and increase cache hit rate for biological transcriptomic data. The perfomance is compared with relational models in MySQL cluster and MongoDB.

Considering the features of business data, the authors in [17] present an optimized HBase table schema focusing on merging detailed information to fit in combination with customer cluster and constructing an index factor scheme to improve the calculation of strategy analysis formulas.

In summary, the above-referenced prior efforts tend to focus on optimizing the table schema, row key design for data fast access, update and query. For our work, we not only provide an innovative row key hierarchical design, but also optimize the default RegionSplitPolicy which goes deep into the HBase architecture. Our goal is to maximally collocate relevant data on same node for further and faster group processing.

3. ENHANCEMENTS TO THE APACHE ECOSYSTEM

The task of processing medical images at scale requires a distributed image processing architecture that is aware of the underlying hierarchical imaging data and its metadata. Our system is based upon the Hadoop framework, which was originally designed for file-system management and distributed processing [9, 13]. We combine Hadoop with Apache HBase, a NoSQL database which implements Google's BigTable [13, 7]. The specific contribution of our work is a novel data storage mechanism that uses the hierarchical structure of imaging studies to collocate data with physical machines. This proposed collocation provides an efficient processing environment in which data do not need to be transferred between machines, thus avoiding network overhead and saturation.

Before we introduce our contributions and to make this paper self-contained, we first provide background information on Apache Hadoops' HBase. We also describe the properties of the DICOM and NiFTI file formats used by the medical imaging community.

3.1 Background on Apache HBase

HBase [24] uses the Hadoop Distributed File System (HDFS) to provide distributed and replicated access to data. Zookeeper [18] handles distributed coordination to maintain the state in HBase cluster. It uses consensus to guarantee common shared state; hence the recommended cluster size is an odd number for cluster leader selection.

The key concepts from the HBase architecture are summarized in Table 1. Briefly, HBase maintains tables, which have a row key that is commonly used as an index, and where data columns are stored with the row key. All data in HBase is "type free," which are essentially in the format of a Byte Array. The table is sorted and stored based on the row key.

The HBase tables are divided into "regions" for distributed storage such that each region contains a continuous set of row keys from the overall table. The data in a region is physically collocated with an HDFS data node to provide data locality, which is performed by an operation called major compaction. As a region size grows above a pre-set physical size threshold, a "RegionSplitPolicy" takes effect and divides the region into smaller pieces. The newly created regions are automatically moved to different nodes for load balancing of the entire cluster. The row key and RegionSplitPolicy are integral to the performance and data retrieval of HBase and Hadoop.

There is no standard for default row key design. Intuitively, the data should be placed as sparse as possible and distributed evenly across various points of the regions in the table. Such a strategy can avoid data congestion in a single region, which otherwise could give rise to read/write hotspots and lower the speed of data updates. Because row keys are sorted in HBase, using randomly generated keys when input the data to HBase can help leverage the data distribution in the table. As shown later, however, such an approach incurs performance penalties for medical imaging applications.

3.2 Background on DICOM and NiFTI

DICOM (Digital Imaging and Communications in Medicine) is the international standard for medical images and related information (ISO 12052). It defines the formats for medical images that can be exchanged with the data and quality necessary for clinical use (http://dicom.nema.org/). It has a hybrid structure that contains regular data (patient/clinical information), multimedia data (images, video). Data inside a DICOM file is formed as a group of attributes [25].

When a patient gets a Computed Tomography (CT) or Magnetic resonance imaging (MRI) scan, for example a patient's brain image, a group of 2-dimensional DICOM images are generated slice by slice. A non-exhaustive set of medical imaging DICOM attributes for the slices include: project, subject, session, scan, where a project is a particular study, a subject is a participant within the study, a session is a

Table 1: HBase architecture key concepts summary

Concept	Comment
Table / HTable	A collection of related data with a column-based format within HBase.
Region	HBase Tables are divided horizontally by row key range into "Regions." A region contains all rows in the table between the region's start key and end key.
Store	Data storage unit of HBase region.
HFile / Storefiles	The unit of Store, which is collocated with a Hadoop datanode and stored on HDFS.
memStore	When write data is uploaded to a HTable, it is initially saved in a cache as memStore. Once the cache size exceeds a pre-defined threshold, the memStore is flushed to HDFS and saved as HFile.
HMaster	HBase cluster master to monitor a RegionServer's behavior for load balancing.Table operator. e.g., create,delete and update a table.
Regionserver	Serves read/write I/O of all regions in a cluster node. When Regionservers collocate with Hadoop datanode, it can achieve data locality. Subsequently, most reads are served by the RegionServer from the local disk and memory cache, and short circuit reads are enabled.
Rowkey	A unique identifier of a row record in table.
Column family	Columns in Apache HBase are grouped into column families.
Column identifier	The member in column family, also called as column qualifier. Multiple column identifiers can be used within one column family.

single imaging event for the subject, and a scan is a single result from the event.

In order to study the entire brain, all 2-dimensional DICOM images should be collected together. Even though medical imaging data is stored as DICOM images, a substantial amount of medical image analysis software are NiFTI-aware (e.g. FSL (http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/), AFNI (https://afni.nimh.nih.gov/afni/), SPM (http://www.fil.ion.ucl.ac.uk/spm/) and Freesurfer (http://freesurfer.net/)). NiFTI is a medical image data format, which is termed as a "short-term measure to facilitate inter-operation of functional MRI data analysis software package," developed and founded by the NiFTI Data Format Working Group (http://nifti.nimh.nih.gov/).

Converting a large group of slices of DICOM images belonging to one patient into a small number of NiFTI format images (many-to-many relationship) is a significant step in medical imaging study. Any processing of DICOM datasets will need to determine which CT/MRI **scan** that slice belongs to and using the **Session** attribute which records when the CT/MRI scan volume is carried out. However, finding a **Session** needs to first know the attribute **Subject** that it belongs to. Finally, the **Project** attribute collects all subjects together. Thus, for medical imaging applications involving DICOM, the following attributes are necessary: $project \rightarrow subject \rightarrow session \rightarrow scan \rightarrow slice$.

As motivated in Section 1, traditional grid computing approaches separate DICOM data storage from computation. To complete a DICOM to NiFTI conversion, DICOM datasets must be copied from a storage archive, submitted to an execution node, processed, synthesized to a result, and results returned to a storage archive. When imaging datasets become massive, the bottleneck associated with copying and ensuring consistency overwhelms the benefits of increasing the number of computational nodes.

Despite using Big data architectures, such as Apache Hadoop, a number of challenges present themselves. For instance, the original DICOM file name is a unique identifier called Global Unique identifier [16]. If the task of interest is storing slice-wise DICOM data within HBase, a naïve approach would be to use the DICOM GUID. Since the GUID is a hash of the data, it will not collocate data together and thus will saturate the network at the time of doing retrieval all DICOM images of a scan volumes. Further, the standard RegionSplitPolicy will randomly assign files with hashed DICOM GUID file name as row keys to regions based on the key and the convenient split point based on region size.

3.3 Modified Row Key Design

To address the challenges explained earlier, we propose a novel row key design for HBase based on the row key design requirements that it must maintain the structure of DICOM comprising the project, subject, session, scan. To maintain this structure, we propose using $< ProjectID >\_< SubjectID >\_< SessionID >\_< ScanID >$ as the identifier with other optional characteristics such as the "slice" appended to this identifier. This is how our collection of images are named in the hierarchical manner. For example, a row key is like $Proj1\_Subj2\_Session3\_Scan4\_Slice5\_example.dcm$, where ".dcm" is the filename extension for DICOM. Since HBase organizes data linearly based on row key, this new strategy will maintain data within a project that is split with a minimal, or just one more than the minimal number of splits across regions as possible, when used in conjunction with the default RegionSplitPolicy supported by HBase.

3.4 Modified RegionSplitPolicy for Medical Imaging

The default RegionSplitPolicy does not have knowledge of the structure of medical imaging studies. Thus, it may split the regions non-ideally such as within a scan or between sessions for a particular subject. These splits cause increased processing time when data needs to be transferred between machines. To overcome these bottlenecks, we propose a novel RegionSpliyPolicy, which has knowledge of the modified row key structure, thus maximizing data co-localization.

In practice, users always select a cohort (set of subjects, sessions) to do the processing. The data under the same subjects or sessions are always processed together, not individually. Specifically, for DICOM conversion, the unit of processing is scan volumes. Thus, it is important to maximally collocate relevant image data under the same level for

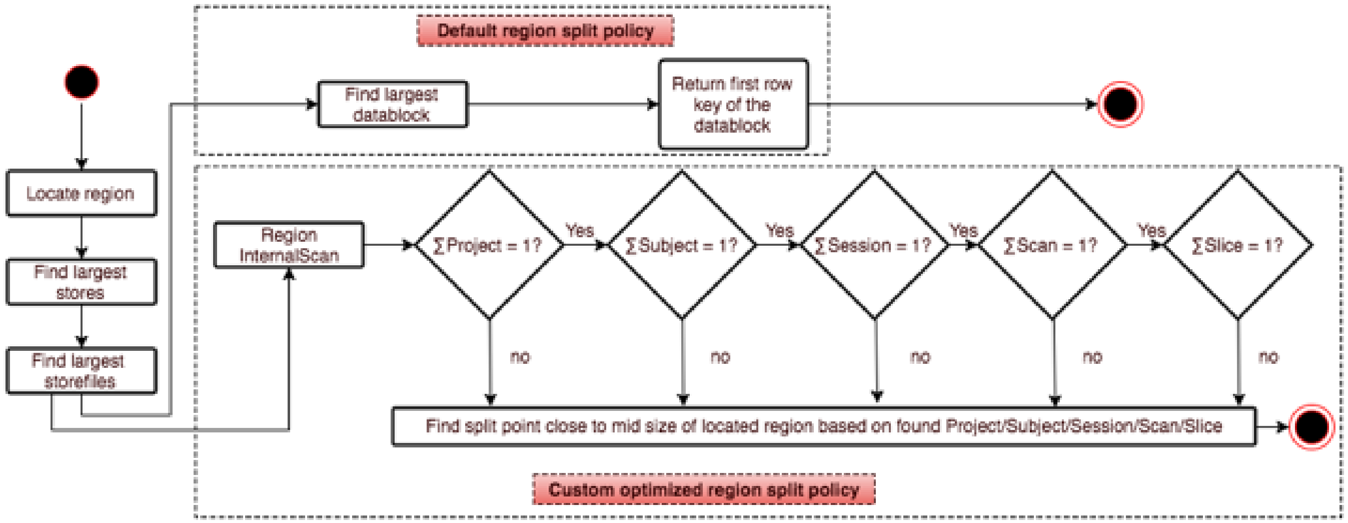
Default region split policy
Locate region
Find largest stores
Find largest storefiles
Find largest datablock
Return first row key of the datablock
Region InternalScan
∑Project = 1?
∑Subject = 1?
∑Session = 1?
∑Scan = 1?
∑Slice = 1?
Yes
no
Find split point close to mid size of located region based on found Project/Subject/Session/Scan/Slice
Custom optimized region split policy

Figure 1: Comparison of the standard RegionSplitPolicy and our custom RegionSplitPolicy. The standard policy splits the data within a region equally based on the data in the region. The custom policy considers the projects, subjects, sessions and scans in the region and makes a split to maximize data co-locality.

further group retrieval and processing/analysis, while reducing the data movement in MapReduce operations pertaining to the DICOM to NiFTI conversion. The details of the MapReduce operation is discussed in Section 4.4.1.

Our new RegionSplitPolicy first considers all row keys in a region. If multiple projects exist in the region, it splits the projects into separate regions. If the region is homogeneously a single project, it finds the highest available level (project, subject, session, scan) in the region on which it can split and balance the data between the new regions. This proposed regionsplit policy relies on the row key design described in Section 3.3. Figure 1 compares the operation of the standard RegionSplitPolicy and the custom one we have developed and evaluated.

In detail, when a region split is triggered, the RegionServer first finds the largest files in the largest stores. The default RegionSplitPolicy defined in HBase finds the first row key of the largest data block in each storefile. This key is called the "midkey" of a region and is decided based on region mid size. Thus, this split point can separate an existing associated imaging dataset into two regions without considering what row keys values in the split region. The newly created two regions will move through the whole cluster for storage balancing.

The challenge for our optimized RegionSplitPolicy is to find a split point based on all row keys of a Region. HBase provides a client API to retrieve data called scan (here, we refer to it as simple_scan). A use can customize the scan to define the range of row keys with which the column family and identifiers need to be retrieved. Users can also set customized filters to refine the query scan. A region has internal attributes that record the value of the start row key and end row key of the region. Since there are no attributes of records for any other row keys in a region except start / end row key, we need to use external ways to retrieve all row keys of a region.

In order to get all keys in a region, two potential ways can be used in traditional HBase: (1) According to start/end key of region, a user specifies a column to scan. The scan is first executed on the entire table, finds the right RegionServer that hosts region from Zookeeper quorum, and retrieves the row key; (2) Use HBase default RowKey filter to customize the scan. However, both approaches are slower compared to our approach described below.

As shown in Figure 1, we are capable of locating the largest storefiles. In this way, we can apply another more advanced HBase scan API (called "Region Internal scan"), which we have found to be 163 times faster than simple_scan on average in our tests to find all hierarchy row keys involved in the region. The Internal scan can directly operate on storefiles located on HDFS without starting a scan from entire table. This gives us all the row keys of a split region. Next, the split point is selected according to the following conditions: (1) it ensures that the maximum related data is collocated in hierarchy, and (2) once we have identified the level of structure which will be the potential point to split, we traverse the candidates and return the point that can most evenly balance the size of two new regions in order to avoid the overhead of so many small regions emerging.

HBase provides PrefixSplitKeyPolicy as one of the default split policy which is designed for grouping rows sharing a fixed length of keys [2]. However, compared to our custom policy, it cannot dynamically group the subjects based on the order of project, subject, session etc based on highest available level (project,subject, session, scan). Namely if there are many projects of a region, we should split rows by $< ProjectID >$; if all row keys start with same project, and there is not only one subject, we should split rows by $< ProjectID > \_ < SubjectID$, so on so forth. So we cannot defined which is the appropriate fixed length for PrefixSplitKeyPolicy.

Another policy we introduce, evaluate and compare with is IncreasingToUpperBoundRegionSplitPolicy [2], which splits a region from a small threshold; with the number of regions increased in a Regionserver, the split size threshold are increasing. It is size-based policy without knowledge of what key values are in a split region.

The observed average run time range to determine one split point using our custom split policy is 28.22-58 ms, and 1.43-1.64 ms for the default HBase split policy with similar CPU usage (19.39% vs. 19.81%), which means the proposed split policy does not involve any substantial overhead when compared with the default one. The increased time in our policy is due to the need to retrieve and analysis all row keys of a region. Despite this one-time initial cost, as we show in our experimental results, the performance improvements are substantial.

3.5 Putting the Pieces Together

Figure 2 presents the overall structure of our modified Apache Hadoop ecosystem focusing particularly on the HBase modifications. HBase resides upon HDFS. Zookeeper monitors the health status of RegionServer. When users create a HBase table, they need to pinpoint the RegionSplitPolicy to HMaster, and the pre-set split policy is automatically triggered once when a Table region needs to be split. Our custom split policy is made the default split policy. The input DICOM is de-identified (for privacy preservation purposes) and is normalized to the hierarchy structure by a local row key generator before storing into HBase.

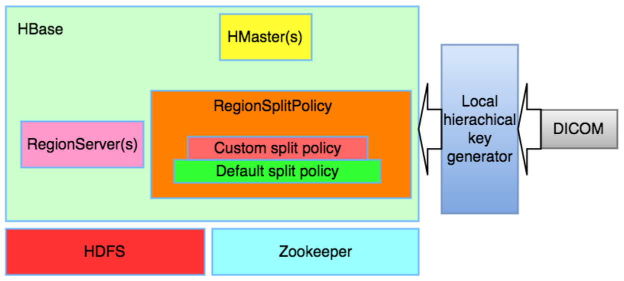

Figure 2: Overall structure of Hadoop / HBase / Zookeeper cluster with proposed custom row key and custom region split policy

4. EVALUATIONAL METHODOLOGY AND EXPERIMENTAL RESULTS

This section presents results of evaluating our Apache Hadoop/HBase modifications and comparing them with default strategies.

4.1 Testing Scenarios

To investigate the performance of our HBase modifications, we evaluated the standard DICOM to NiFTI file format conversion using three test scenarios using HBase and Hadoop and one with Network Attached Storage (NAS) as follows.

1. **Scenario: "Naïve HBase"** – The MD5 hash was used in place of the DICOM GUID because GUIDs were removed during the de-identification process associated with data retrieval. Using MD5 hash key value meets original intentional HBase key design for reduce hot-spot of table read/write. The DICOM files are distributed to all HBase regions, and we use an additional table to record the hierarchy structure of a scan dataset. We test using a random key, and MD5 hash of the data, as the key in HBase. With this comparison, we test the native capabilities of Hadoop and HBase without any of our proposed advances.

2. **Scenario: "Custom Key/Default Split HBase"** – This scenario evaluates the custom key grouping and ordering of the DICOM file logically and physically in HBase by our custom key value prefix. When a HBase region exceeds a pre-defined size, we use the default split policy to split a region into two child regions without considering the key values of the split region as introduced in Section 3.3. In this case, the files belonging to the same project, subject, session, scan are distributed into two different regions. The two regions may move to different cluster nodes, and the replication of both regions may also be placed on random Hadoop datanodes. When retrieving all files of a cohort (i.e., a set of scan volumes) for further processing, MapReduce job dispatches computations to nodes that contain the datasets of interest. When no single node contains all requested data for a single job (either due to a large request or local storage scarcity), the minimal necessary data will be retrieved over network. So we test our proposed row key with the default RegionSplitPolicy.

3. **Scenario: "Custom Key/Custom Split HBase"** – Our custom RegionSplitPolicy has the capabilities to maximally collocate relevant data in the same group, with the order of project, subject, session and scans. We test our complete design with our proposed row key and custom RegionSplitPolicy and compare it with Custom Key/Default Split HBase to see how data retrieval matters in MapReduce. Theoretically, this approach involves less data collection and movement via the network than the other two HBase methods and makes processing faster.

4. **Scenario: "Grid Engine NAS"** – Traditional grid computing approaches separate data storage from computation. As a comparative method, we use a traditional Sun grid engine (SGE) to distribute portable bash script (PBS) jobs to computational nodes accessing data from a Network Attached Storage (NAS) device.

4.2 Hardware

Twelve physical machines were used consisting of 108 cores of AMD Operon 4184 processors, 40 cores of Intel Xeon E5-2630 processors and 8 cores of Intel Xeon W3550 processors running Ubuntu 14.04.1 LTS (64 bit). At least 2 GB RAM was available per core. In total, 190 GB of storage was allocated to HDFS and a Gigabit network connected all of machines. Each machine was used as a Hadoop Datanode and HBase RegionServer for data locality. All machines were also configured using the Sun Grid Engine (Ubuntu Package: gridengine-* with a common master node). NAS was provided via CIFS using a Drobo 5N storage device (www.droboworks.com) with a 12 TB RAID6 array.

4.3 Data and Processing

To evaluate the test scenarios, 9,910,000 DICOM files from clinical computed tomography scanners corresponding to 410 subjects and 8120 scan volumes were retrieved in de-identified form under IRB approval from a study on traumatic brain injury. The processing system for each scan applied a command line program to retrieve the data from storage (see test scenarios in Section 4.1) and convert the DICOM files to NiFTI using dcm2nii (https://www.nitrc.org-/projects/dcm2nii/). We performed tests with the subsets of the data with different number of scans (See Table 2) to assess the scalability of each proposed system and relative overhead versus processing load.

Table 2: DICOM datasets size info

Datasets	**Total Scan size (GB)**
104	7.16
186	10.93
294	19.05
407	27.55
497	34.12
606	41
718	47.14
812	53.01
1624	106.02
2436	159.03
3248	212.04
4060	265.05
4872	318.06
5684	371.07
6496	424.08
7308	477.09
8120	530.1

4.4 Apache Hadoop/HBase Experimental Setup

In our experimental setup, the Sun grid engine does the balancing and makes sure that the jobs ran as soon as space was available within the specified node list when processing is executed on a traditional grid [12]. For Hadoop scenarios, MapReduce is a programming model and an associated implementation for processing large datasets in the Hadoop ecosystem [9]. YARN is used for resource (CPU/Memory) allocation and MapReduce job scheduling [28]. We use the default YARN capacity FIFO (First in First Out) scheduler, which aims at maximizing the throughput of cluster with capacity guarantees when the cluster is being shared with multi-tenant.

The software tools to generate row keys from DICOM data were implemented in open source. The custom region split policy was implemented as a Hadoop extension class. All software is made available in open source at NITRC project Hadoop for data collocation (http://www.nitrc.org/projects/hadoop_2016/). Manual inspection of region stores was used to verify data collocation under multiple configurations of Hadoop Datanodes to ensure that the desired data collocation and region splits were occurring.

4.4.1 *MapReduce Setup for HBase approach*

The MapReduce model should complete two main tasks: data retrieval from HBase and data processing (DICOM to NIFTI conversion). The Map phase always tries to ensure that the computation tasks are dispatched where data resides, and those tasks are vividly called data-local maps. A compromise scenario is when data is not on the local node where the running map task is located, but at least the data are on the same rack, and those maps are rack-local maps. In the Reduce phase, the output,$< key, value >$ pairs from Map phase are to be shuffled/sorted and sent to random cluster nodes.

If data retrieval is done in the Map phase while processing in the Reduce phase, then the computation and data can be on different nodes. A potential way to execute the processing is remote access (i.e. SSH) where data originally locates and applies processing. SSH limitation may occur, however, and block further connection, and as a result the processing cannot be fully completed. If both data retrieval and processing occurs in the Reduce phase, namely, each reduce task collects all row keys related with one scan volume, then downloads and collect DICOM files from HBase according to the keys, and finally executes the conversion in Reduce phase. In this way, network congestion occurs when the node that holds the Reduce task may not have all needed DICOM files. Since those DICOM files are aggregated in a same Region / node owing to proposed custom split policy, so Reduce task has to retrive all datasets through network and leads to congestion.

Thus, these approaches break our main goal for data collocation with Hadoop and HBase with minimum data movement. As a result, for good use our proposed Hadoop enhancements with data collocation in the context of the hierarchical key structure, data retrieval and processing occur in the Map phase and the Reduce phase is a no-op for our application.

In a traditional "word count" example, the input of MapReduce is a HDFS folder. The input folder is split into several pieces based on the files in the selected folder. Then each piece starts a map task with $< key, value >$ pair, the input Map Key is file names and input Map value is file content. However, this approach is not practical in HBase. The HBase region has a corresponding folder on HDFS, and all data stores/hfiles in this region are placed in the region. When the region collocates to a Hadoop datanode to achieve data locality, all data store/hfiles are compacted to a giant file, which means that a traditional MapReduce like word-count strategy cannot split a input HBase folder for further processing.

Figure 3 shows the modified work-flow. HBase provides a default API for running HBase-oriented MapReduce. The input of the MapReduce is a HBase-scan, which represents an interval of consecutive table row key values of a selected column. The HBase-scan is split based on relevant regions, and the input $< key, value >$ pairs are values about row keys of a region and the content of the specified column. In short, if the input HBase-scan occurs across n regions, then only n map tasks are generated. The challenge for traditional HBase-oriented MapReduce for DICOM is there are usually more than one datasets of DICOM files under the same scan in a region. So we refined the above approach to specify input of MapReduce to be a selected cohort of scan volumes, and the number of Map tasks is based on the number of scans. The Map Phase first retrieves the data from HBase and stores DICOM files to local node. Once done, it converts the DICOM files to NiFTI using dcm2nii as presented in Figure 3. For fair comparison between Hadoop methods and approach on NAS, additional steps such as uploading the NiFTI result to HBase is not launched.

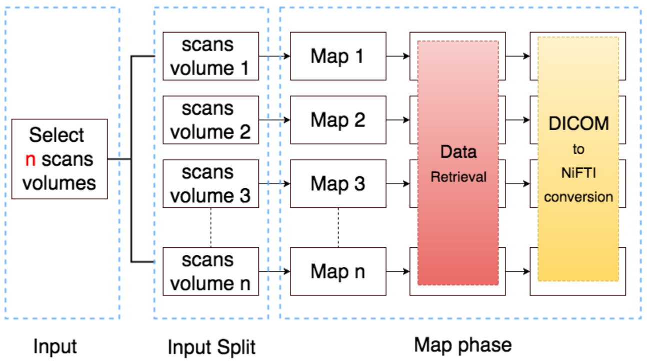

Figure 3: Custom HBase oriented MapReduce basing on input selected groups of scan volumes

4.4.2 *Guidelines used for Scaling Hadoop / HBase Cluster*

Scalability is one of the most important properties for Cloud usage. We test and scale our clusters for studying intrinsic scalability performance. The following summarizes how we scaled the Hadoop / HBase cluster step by step.

- For scaling down, RegionServer should first be gracefully stopped [2], and relationship of data collocation between Datanode and RegionServer are no longer exists. Then major compaction on the affected data from stopped RegionServer must be applied to collocate to the rest of the cluster [2]. When all data-locality is achieved again, decommission the Datanode and rebalancing of the cluster is performed. If decommission order is reversed, redundant replications are to be stored into HDFS which exponentially decreases the available size of the Hadoop cluster.

- For scaling up, a new Hadoop Datanode must be commissioned first and then a new HBase RegionServer is added, followed by a major compaction to achieve data locality. If there is no Datanode, adding a new RegionServer can collocate to nothing, which makes reverse commissioning order no sense.

4.5 Results of Data Transfer Latency

First, we evaluated the latency in retrieving imaging data in each of the four scenarios. Table 3 shows average latency for all datasets. For naïve Hadoop, we retrieved data to a random node since the data were not collocated. For custom key / standard split, we retrieved the data to the machine which contained the first element in the scan. For custom key / custom split, we retrieved the data to the machine where the data were located entirely. For Grid Engine NAS, we retrieved the data from the NAS to a local machine serially (i.e., with one core in use).

The naïve Hadoop strategy performed markedly worse than the other methods because it needs to open and close connections with multiple other machines in order to download the data, and the initialization and setup of each ZooKeeper connection involves overhead. Using the NAS with a single

Table 3: Latency results in seconds for each of the four test scenarios.

Approach	Grid Engine NAS	Naive HBase	Custom key/ Standard split HBase	Custom key/ Custom split HBase
Latency(s)	4.76	19.02	3.29	2.56

connection is relatively effective since the data are coming from one fixed location and there is low overhead in opening and closing connections. In comparing the default split policy to our proposed policy, we see an improvement in average performance. Any increase comes from the cases where scans are split between machines and thus data needs to be retrieved from other locations on the network.

4.6 Data Processing Throughput for DICOM to NiFTI Conversion

Each of the four scenarios executed a DICOM to NiFTI conversion as described in Section 4.3. Figure 4A presents an analysis of throughput. The Grid Engine NAS performed the worst (fewest datasets per minute, longest run times) across all dataset sizes. In all scenarios, the NAS device saturated at 20 MB/s (approximately 18 datasets per minute) throughput despite the gigabit network access. This was likely due to numerous small files that are generated with “classic” DICOM scanning as direct read/write to the NAS device demonstrated substantively higher performance.

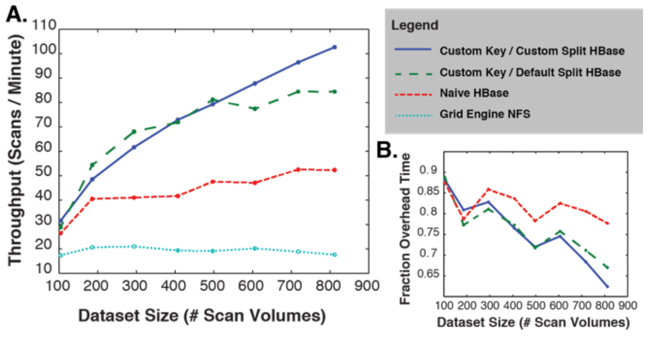

Figure 4: Throughput analysis for each of the test scenarios. (A) presents the number of datasets processed per minute by each of the scenarios as a function of the number of datasets selected for processing. (B) shows the fraction of time spent on overhead relative to the number of datasets.

The naïve HBase approach scaled better than the NAS approach with a throughput ranging from 31 MB/s (with 104 datasets) to 58 MB/s (with 718-812 datasets). The performance leveled off at 52 datasets/minute for a factor of almost three-fold improvement over NFS. The custom key / default policy HBase approach performed even better with a throughput of 34 MB/s (with 104 datasets) to 94 MB/s (with 718-812 datasets). The custom key / custom policy HBase approach further increased throughput performance from 37 MB/s (with 104 datasets) to 114 MB/s (with 812 datasets).

The naïve method’s performance increases flatly because of uncertainty in the placement of data loading. It performs better than processing on the NAS device because not all data needs to be retrieved from other node; some of the files are placed on same node with Map computation in most cases. On the other hand, the custom key / custom policy HBase involves smaller data movement with better performance rather than the custom key / default policy HBase, both of whose processing are executed within most data-local map and a few rack-local map according to YARN allocation.

4.6.1 *Throughput Upper-bound*

Figure 4-A illustrates the processing on NAS device, which saturates the Gigabit network. The HBase approach does not incur as much network congestion because most map tasks are data-local or rack-local. Thus, we were not able to observe any perceived network-imposed limitations even until 812 datasets. The upper limit on the throughput stems from other overheads in the framework, which we address in the paper. This is further verified in Section 4.7.

Consequently, to identify the upper limit on the throughput of our system, we tested our system for more number of datasets. Figure 5 presents the result of processing scans per minute with more datasets according to Table 2. Our proposed methods performs better than custom key / default policy HBase initially (with 407–4,060 datasets, 34.12–265.05 GB, respectively), and it reaches its throughput upper bound with 124.5 MB/s (with 4,060 datasets). Thereafter, both approaches perform basically the same. The throughput upper bound for custom key / default policy HBase is 120.5 MB/s (with 4,872 datasets). Since network is not a factor, we conclude that to obtain even higher throughput, we will need to scale the hardware by adding more cores since the number of cores is the limitation factor.

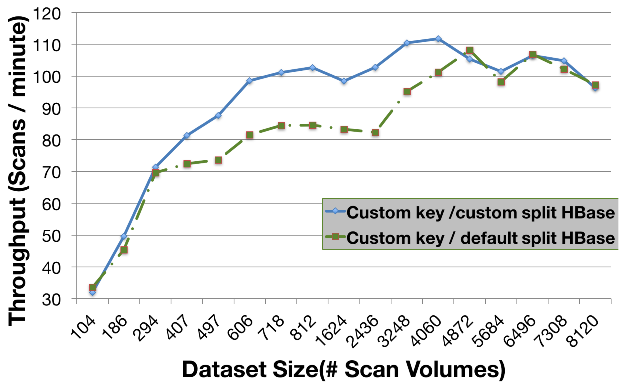

Figure 5: Throughput analysis for finding upper-bound of scenarios custom key / default policy HBase and custom key / custom policy HBase

4.6.2 *Overhead Considerations with the Hadoop Framework*

The computing grid had 108 cores available. Therefore, up to 108 jobs could run simultaneously in any of the test scenarios. With the three Hadoop scenarios, we have logs of both the time spent within each job on the compute node (including time to establish data connection, retrieve the data, and clean up the connection) and the actual wall time.

For each of the Hadoop scenarios, we computed the average actual time spent executing the processing (including data retrieval), which ranged from 22 s to 35 s. For each of the data submission tasks, we can identify the minimum number of jobs that would need to run in serial by dividing the number of scan volumes by the number of cores. The fastest time that the Hadoop scheduler could run the jobs is the length of the serial queue times the job length, but in all cases the actual wall time exceeded this value. We define the overhead time as the difference between the actual wall time and the theoretical minimum time.

The ratio of overhead time to total time is shown in Figure 4B. Fitting a linear analysis to each of the three scenarios, shows that the naïve HBase strategy had a marginal penalty of 1,003 ms per additional dataset. The custom key / default split policy reduced the overhead penalty to 547 ms per additional dataset. Finally, the custom key / custom split policy resulted in 398 ms per additional dataset.

4.6.3 *Overhead lower-bound*

Similar to Section 4.6, the overhead of Hadoop scenarios have not reached the bottom line within 812 datasets. Figure 6 shows ratio of overhead in total processing time with multi-datasets. Both overhead values linearly decrease and then become steady at 33%. Finally, the Custom key / default split policy reaches in 159 ms as lower-bound and the Custom key / default split policy performs a bit better with 138 ms. When the size of datasets is small, the time for establishing data connection, retrieving the data, and cleaning up the connection dominates total time compared with data processing. When the framework reaches the upper limit of cluster cores treating processing capability, the overhead is balanced.

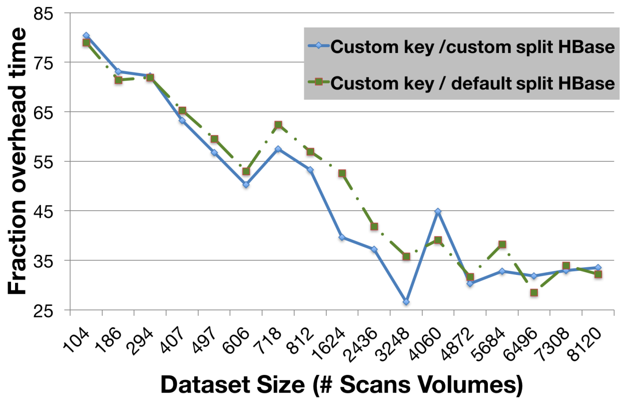

Figure 6: The fraction of time spent on overhead relative to the number of datasets for finding lower-bound of scenarios custom key / default policy HBase and custom key / custom policy HBase

4.7 Evaluating the Scalability of the Framework

We wanted to understand how does the scale of the cluster impact performance. Thus, we experimented by linearly decreasing the size of the cluster and observe if the performance decreased in similar manner. In our experiments, each machine acted as a Hadoop Datanode and HBase RegionServer for data locality as introduced in Section 4.2. The order of decommisioning of Datanode and RegionServer is important when scaling the size of the cluster. Custom key with default and custom split policy are compared on scaled cluster (5-10 Hadoop/HBase nodes). Decreasing the size of the cluster can linearly increase the total time with processing 5,684 datasets, which is presented in the trends of Figure 7.

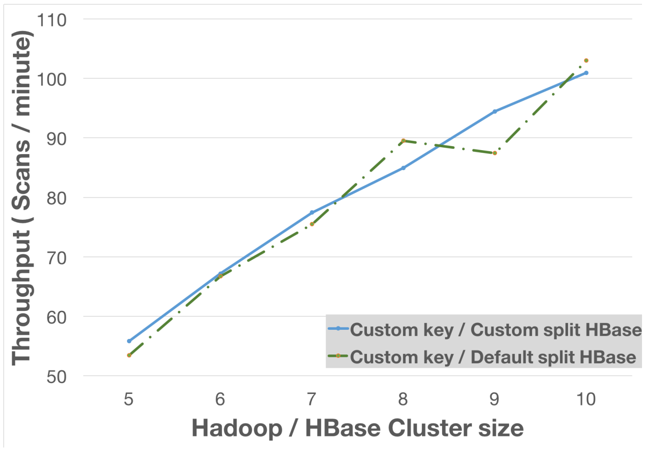

Figure 7: Throughput analysis for Hadoop scenarios with different size of cluster

Based on the previous discussion, we can conclude that the Hadoop scenario performance is not limited by the network bandwidth but by the total available CPU cores and memory. Thus, scaling up the size of cluster can increase high performance computing capability for medical imaging processing in an affordable local/cloud-based commodity grid.

5. CONCLUSIONS

Big data in medical imaging offer an opportunity to study specific control populations (age / sex / demographics / genetics) and identify substantive homogeneous sub-cohorts so that one may understand the role that individual factors play in treatment response. Billions of magnetic resonance imaging (MRI) and computed tomography (CT) images on millions of individuals are currently stored in radiology archives [3]. These imaging data files are estimated to constitute one-third of the global storage demand [11], but are effectively trapped on storage media.

The medical image computing community has heavily invested in algorithms, software, and expertise in technologies that assume that imaging volumes can be accessed in their entirety as needed (and without substantial penalty). Despite the promise of big data, traditional MapReduce and distributed machine learning frameworks (e.g., Apache Spark) are not often considered appropriate for “traditional” / “simple” parallelization. Herein, we demonstrate that Hadoop MapReduce can be used in place of a PBS cluster (e.g., Sun Grid Engine). Moreover, with our approach even a naïve application of HBase results in improved performance over NAS using the same computation and network infrastructure.

We present a row key architecture that mirrors the commonly applied Project / Subject / Session / Scan hierarchy in medical imaging. This row key architecture improves throughput by 60% and reduces latency by 577% over the naïve approach. The custom split policy strongly enforces data collocation to further increase throughput by 21% and reduce latency by 29%. With these innovations, Apache Hadoop and HBase can readily be deployed on commodity network to address the needs of high throughput medical image computing.

As implied by the trends in Figure-4, the benefits of distributing computation with storage increase with larger datasets. Exploration of the asymptotic performance limits is of great interest, but beyond the scope of this initial presentation that illustrates meaningful gains on problems of widely applicable scale. The optimization of characterization of these approaches on heterogeneous grid is an area of great possibility. In particular, the Apache Hadoop YARN scheduler could be further optimized to exploit intrinsic relationships in medical imaging data.

Several broader domains have the capability to apply our proposed work. Gene data have many different styles with diverse attributes. Genes with similar expression patterns must be collocated for group analysis since genes that behave similarly might have a coordinated transcriptional response, possibly inferring a common function or regulatory elements [4]. Thus, genes data group/hierarchy storage, retrieval and analysis is applicable by our framework.

Another scenario where our work is applicable includes Satellite data/image processing on data about earth surface, weather, climate, geographic areas, vegetation, and natural phenomenon [15], which can be studied according to day-based, multiple-day-based, or seasonal-based [14]. As a result, time-oriented hierarchical structure can help group the data from the satellite for further processing. similarly, Internet of things collect data from various facilities like sensors. According to the sensors' supervision area, a component hierarchy-based data collection can be implemented. For instance, high-speed train fault and repair prediction is applied before a train runs [29]. Analyzing mass historical data from a group of Electric Multiple Unit (EMU) of a train's components has potential to be implemented in our framework.

Acknowledgments

This work was funded in part by NSF CAREER IIS 1452485. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NSF.

6. REFERENCES

[1] Apache Hadoop Project Team. The Apache Hadoop Ecosystem. http://hadoop.apache.org/.

[2] Apache HBase Team. *Apache hbase reference guide.* Apache, version 2.0.0 edition, Apr. 2016.

[3] AT&T. Medical imaging in the cloud. Technical Report AB-2246-01, AT&T, July 2012.

[4] T. Barrett and R. Edgar. [19] gene expression omnibus: Microarray data storage, submission, retrieval, and analysis. *Methods in enzymology*, 411:352–369, 2006.

[5] T. Bednarz, D. Wang, Y. Arzhaeva, R. Lagerstrom, P. Vallotton, N. Burdett, A. Khassapov, P. Szul, S. Chen, C. Sun, et al. Cloud Based Toolbox for Image Analysis, Processing and Reconstruction Tasks.

In *Signal and Image Analysis for Biomedical and Life Sciences*, pages 191–205. Springer, 2015.

[6] Y. Chabane, L. d'Orazio, L. Gruenwald, B. Mohamad, and C. Rey. Medical Data Management in the SYSEO Project. *ACM SIGMOD Record*, 42(3):48–53, 2013.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[8] S. Chen, T. Bednarz, P. Szul, D. Wang, Y. Arzhaeva, N. Burdett, A. Khassapov, J. Zic, S. Nepal, T. Gurevey, et al. Galaxy+ Hadoop: Toward a Collaborative and Scalable Image Processing Toolbox in Cloud. In *Service-Oriented Computing–ICSOC 2013 Workshops*, pages 339–351. Springer, 2013.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[10] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens. Mapping Brain Activity at Scale with Cluster Computing. *Nature methods*, 11(9):941–950, 2014.

[11] Frost and Sullivan. U.S. Data Storage Management Markets for Healthcare, Nov. 2004.

[12] W. Gentzsch. Sun grid engine: Towards creating a compute power grid. In *Cluster Computing and the Grid, 2001. Proceedings. First IEEE/ACM International Symposium on*, pages 35–36. IEEE, 2001.

[13] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *ACM SIGOPS operating systems review*, volume 37, pages 29–43. ACM, 2003.

[14] N. Golpayegani and M. Halem. Cloud computing for satellite data processing on high end compute clusters. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 88–92. IEEE, 2009.

[15] D. Gorgan, V. Bacu, T. Stefanut, D. Rodila, and D. Mihon. Grid based satellite image processing platform for earth observation application development. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2009. IDAACS 2009. IEEE International Workshop on*, pages 247–252. IEEE, 2009.

[16] V. Hernández et al. Bridging clinical information systems and grid middleware: a medical data manager. In *Challenges and Opportunities of Healthgrids: Proceedings of Healthgrid 2006*, volume 120, page 14. IOS Press, 2006.

[17] S. Hong, M. Cho, S. Shin, C. Seon, S. Song, et al. Optimizing hbase table scheme for marketing strategy suggestion. In *2016 8th International Conference on Knowledge and Smart Technology (KST)*, pages 313–316. IEEE, 2016.

[18] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[19] S. Jai-Andaloussi, A. Elabdouli, A. Chaffai, N. Madrane, and A. Sekkaki. Medical Content-based Image Retrieval by using the Hadoop Framework. In *20th International Conference on Telecommunications (ICT), 2013*, pages 1–5. IEEE, 2013.

[20] Q. Li, Y. Lu, X. Gong, and J. Zhang. Optimizational method of hbase multi-dimensional data query based on hilbert space-filling curve. In *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2014 Ninth International Conference on*, pages 469–474. IEEE, 2014.

[21] T. Ma, X. Xu, M. Tang, Y. Jin, and W. Shen. MHBase: A Distributed Real-Time Query Scheme for Meteorological Data Based on HBase. *Future Internet*, 8(1):6, 2016.

[22] Y. Ma, J. Rao, W. Hu, X. Meng, X. Han, Y. Zhang, Y. Chai, and C. Liu. An efficient index for massive iot data in cloud environment. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2129–2133. ACM, 2012.

[23] D. Markonis, R. Schaer, I. Eggel, H. Müller, and A. Depeursinge. Using MapReduce for Large-scale Medical Image Analysis. *arXiv preprint arXiv:1510.06937*, 2015.

[24] C. McDonald. An in-depth look at the hbase architecture, 2015.

[25] B. Mohamad, L. d'Orazio, and L. Gruenwald. Towards a Hybrid Row-column Database for a Cloud-based Medical Data Management System. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, page 2. ACM, 2012.

[26] T. S. Soares, M. A. Dantas, D. D. De Macedo, and M. A. Bauer. A Data Management in a Private Cloud Storage Environment Utilizing High Performance Distributed File Systems. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2013 IEEE 22nd International Workshop on*, pages 158–163. IEEE, 2013.

[27] B. Tripathy and D. Mittal. Hadoop based Uncertain Possibilistic Kernelized C-means Algorithms for Image Segmentation and a Comparative Analysis. *Applied Soft Computing*, 2016.

[28] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[29] B. Wang, F. Li, X. Hei, W. Ma, and L. Yu. Research on storage and retrieval method of mass data for high-speed train. In *2015 11th International Conference on Computational Intelligence and Security (CIS)*, pages 474–477. IEEE, 2015.

[30] S. Wang, I. Pandis, C. Wu, S. He, D. Johnson, I. Emam, F. Guitton, and Y. Guo. High dimensional biological data retrieval optimization with nosql technology. *BMC genomics*, 15(8):1, 2014.

[31] C.-T. Yang, W.-C. Shih, L.-T. Chen, C.-T. Kuo, F.-C. Jiang, and F.-Y. Leu. Accessing Medical Image File with Co-allocation HDFS in Cloud. *Future Generation Computer Systems*, 43:61–73, 2015.