Parallel Computing 69 (2017) 63–77

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Hybrid storage architecture and efficient MapReduce processing for unstructured data

Weiming Lu*, Yaoguang Wang, Jingyuan Jiang, Jian Liu, Yapeng Shen, Baogang Wei

College of Computer Science, Zhejiang University, Hangzhou, P.R. China

ARTICLE INFO

Article history:
Received 25 October 2013
Revised 23 July 2017
Accepted 30 August 2017
Available online 4 September 2017

Keywords:
Hybrid storage
Partitioning strategy
MapReduce-based data processing

ABSTRACT

As we are now entering the era of data deluge, how to efficiently manage these massive data is becoming a great challenge, especially for the exponentially growing unstructured data, which is far more than structured and semi-structured data. However, unstructured data is more complex for its variety. That is to say, different types of unstructured data have different file size, type and usage, which need different storage and processing for high efficiency. In this paper, we propose a hybrid storage architecture to store the pervasive unstructured data. This hybrid architecture integrates various kinds of data stores within a unified framework, where each type of unstructured data can find its suitable placement policy and it is transparent to users. In addition, we present several partitioning strategies based on the unified framework, which are beneficial to the MapReduce-based batch processing for these unstructured data. The experiments demonstrate that it is possible to build an efficient and smart system through the hybrid architecture and the partitioning strategies.

1. Introduction

In the past few years, the explosion of data has made a great impact on our lives. Surprisingly, about 80 percent of the data would be unstructured according to Gartner's report in 2010, which defined *unstructured data* as content that does not conform to a specific, pre-defined data model, such as images, videos and documents. In our CADAL digital library,[1] more than 80% of data among 2.5 million digitized books is unstructured, such as digitized book files, charts and illustrations. IDC forecasts that the amount of data will be 35ZB in 2020, so it brings a great challenge to massive data storage and processing. In order to seize the opportunity and distill value from big data, it is of great importance to manage these ever-growing data efficiently, especially the dominant unstructured data.

In addition to the variety, unstructured data has some other implicit properties, which make the unstructured data management even more difficult.

Firstly, unstructured data does not conform to a pre-defined data model, so relational databases are not suitable for managing these data. Examples of unstructured data include: images, videos, audio files, web pages, social media messages,

* Corresponding author.
E-mail address: luwm@zju.edu.cn (W. Lu).
[1] http://www.cadal.zju.edu.cn.

http://dx.doi.org/10.1016/j.parco.2017.08.008
0167-8191/

and many other kinds of business documents, and they are unlike *relational data*, which can be easily mapped into predesigned fields in a database. Therefore, we have to manage the unstructured data as files.

Secondly, unstructured data may be associated with some structured metadata, such as the caption, size and created time of an image. Besides, some valuable information like low-level features and high-level semantic information should be analyzed and extracted from the unstructured data for further usage. Take multimedia information extraction as an example. After loading image datasets, several algorithms could be executed for feature extraction, object detection or scene understanding. That is to say, this is a write-once-read-many scenario, and these valuable information for the unstructured data may evolve gradually along with the data analysis procedure. Therefore, relational databases are still not suitable for managing this valuable information. Moreover, some information like low-level features may be processed in a batch way individually, such as building a high-dimensional index for images with its color feature. Thus, column based storage is more suitable for these data.

Thirdly, many storage systems are developed with their specific objectives, and different unstructured data has different characteristics, which will influence the data management. For example, GFS (Google File System) [1] and HDFS (Hadoop File System) [2] are suitable for huge files, and they are unwieldy to manage billions of approximately KB-sized files such as images. Although HDFS provides several solutions to support small file management, such as *Hadoop Archive* and *Sequence File*, some drawbacks still exist for real applications. For example, *Hadoop Archive* and *Sequence File* can pack HDFS files into a big file, which would makes the data storage efficient, but we cannot locate a certain file in the system quickly, which is the common requirement for photo-sharing websites. In addition, when a large number of small files are packed into a few large files, it could lead to an uneven data distribution, which will harm the performance of unstructured data analysis on the MapReduce framework. Therefore, we have to use multiple stores to manage various types of unstructured data.

In order to address these challenges, in this paper, we proposed a hybrid storage architecture with multiple stores for managing various types of unstructured data efficiently. In addition, in order to efficiently excavate the value of the unstructured data, we proposed several optimal partitioning strategies for MapReduce-based batch processing [3] on the hybrid storage architecture according to the characteristics of different data stores. The partitioning strategies can save IO and improve the efficiency of large-scale data analysis processing.

In summary, the main contributions of this paper are as follows:

- We firstly introduce a unified data model for the unstructured data management. Then, based on the characteristics of different data stores, we proposed a hybrid storage architecture with multiple stores for efficiently managing various types of unstructured data.
- We developed several partitioning strategies for MapReduce-based batch processing on the hybrid storage architecture, including *Key Range based Partitioning Strategy, Group based Workload Balanced Partitioning Strategy* and *Group based Execution Time Balanced Partitioning Strategy*.
- We verify the effectiveness of our hybrid storage system and the partitioning strategies with several empirical experiments.

The rest of the paper is organized as follows. We present the hybrid storage architecture in Section 2 at first, and then introduce the efficient batch processing in Section 3. Following that, we discuss our experimental results in Section 4. Finally, we review related work in Section 5 and conclude our paper in Section 6.

2. Hybrid storage architecture

In this section, we first describe a generalized data model which is suitable for the unstructured data, and then present the overview of the hybrid storage architecture. Finally, we investigate the performance of several data stores and introduce how to place data in the unified storage framework.

2.1. Data model

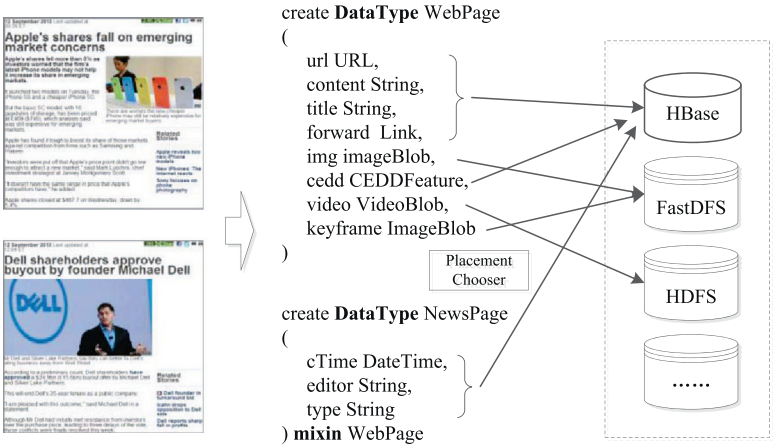
In order to store and process unstructured data in a unified manner, we propose a generalized data model, where every instantiated unstructured data is represented as a data object called *UObject*. UObject contains one or more *Feature*s for its structured metadata and extracted information. For example, an image with its attributes (e.g. file size, image size and file format) and features (e.g. color feature, texture feature and semantic feature) can be represented as an UObject, and the attributes and features of the image are all represented as *Feature*s. Each UObject has a unique data type (*DataType*) and each feature also has its feature type (*FeatureType*). For example, web pages crawled from news sites have a common data type, e.g. *NewsPage*.

DataType abstraction can describe one type of unstructured data, such as *Image* for images and *NewsPage* for a special type of web pages. Users can define a customized DataType which is inherited from another DataType by *mixin* clause. Thus, the newly-created DataType mixes its parent's features into its own. An instance of a DataType is a *UObject*, which is identified by an *ID*. ID is either generated randomly (e.g. UUID) yet guaranteeing uniqueness or encoded by the values of user-specified feature set which is used for indexing UObjects.

FeatureType is declared by primitive value types or user-defined types. We provide many built-in primitive value types, such as Integer, Double, String, URL, DateTime, Vector, Matrix, Blob and Link. Users are also recommended to define feature

```
create FeatureType  ImageBlob Blob
create FeatureType  VideoBlob Blob
create FeatureType  CEDDFeature Vector
Insert into NewsPage (cTime, url, title, img) values ("2013/09/12","http://xxxx.html", "Apple's
shares fall on…", @"D:/tmp/1.jpg")
```

(a)



(b)

Fig. 1. (a) partial usage of modeling language and (b) a simple example with possible storage mapping.

types. In this paper, we exploit Blob type to abstract the unstructured data and define feature types for different types of unstructured data, such as web pages, images and videos. Fig. 1 shows partial usage of modeling languages and an example of data model and possible storage mapping.

In the figure, we can observe that (1) The UObject in real application may contain several types of Features, which may be stored in different stores. For example, url, content and title of a web page may be stored in HBase [4] which provides Bigtable-like structured storage, and the images in web pages may be stored in FastDFS[2] [5] which aims at small files management, while the videos may be stored in HDFS. (2) Features can be extracted from the unstructured data latterly. For example, CEDD (Color and Edge Directivity Descriptor) feature can be extracted from images with a MapReduce procedure.

2.2. Hybrid storage architecture

Fig. 2 shows the hybrid storage architecture. Users store and access their data using the object-based modeling language. The hybrid storage layer (HSL) is responsible for receiving users' requests and providing a unified storage framework over different kinds of data stores.

The hybrid storage layer maintains the connection states using the *client communications manager*, translates the statement (modeling language) into a data operation request and forwards it deeper to be processed. If the request is relevant to data definition, such as FeatureTypes and DataTypes, the *metadata manager* handles it and returns directly. Otherwise, the data manipulation request on storage decision is further routed to the *hybrid storage manager*, and the *statistics collector* builds statistics (e.g. histograms) in parallel. More importantly, the distribution of unstructured data in terms of different kinds of data stores is collected to optimize the following data processing in Section 3.3. Both metadata and statistics are stored in a *Metastore*, which is implemented by HBase in our solution and other databases are easy to replace it.

The hybrid storage manager plays a critical role in the HSL. Only it knows the storage decisions that where the DataObjects are placed. The *storage mapping* component is responsible for mapping features to relational databases since multiple databases are allowed to co-exist. Administrator is able to specify the mapping rules through scripts or implementing a pluggable interface. In addition, the *storage optimizer* is used for choosing the best data store to place data. In this paper, we focus on unstructured data storage management, while structured data is placed via storage mapping. Subsequently, the *physical storage controller* is used for connecting the selected data store and executing physical data operations. Notice that, the holistic HSL is extensible to integrate new data stores by adding corresponding adaptors into the physical storage controller.

[2] http://code.google.com/p/fastdfs/.

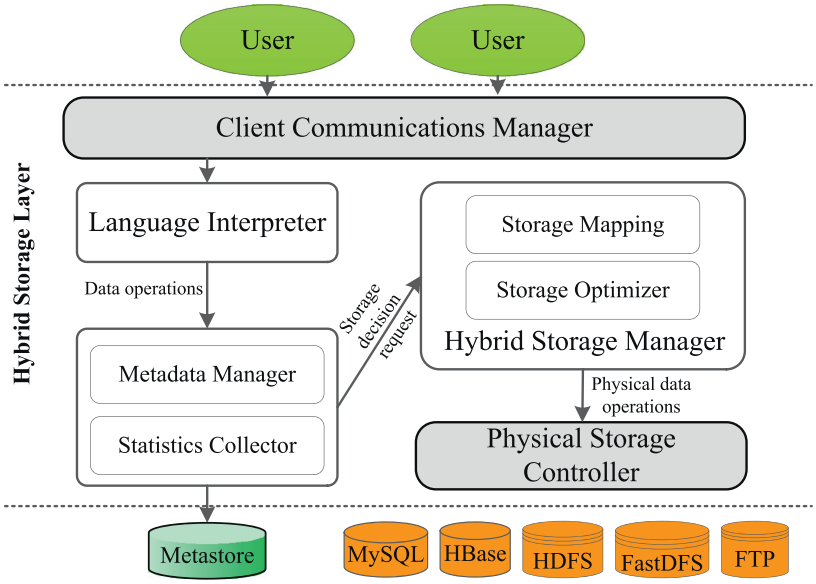


Fig. 2. Overview of the hybrid storage architecture.

2.3. Data stores

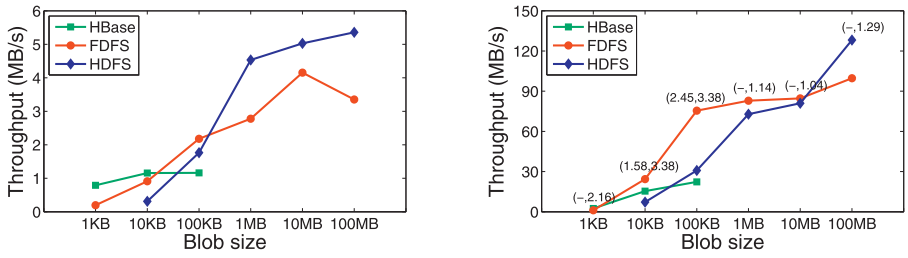
Various data stores have quite different implementations and strengths. It is challenging to determine which one is suitable for a practical production system. There are many factors that impact the decision making, such as scalability and reliability, purchasing cost and usage convenience, and so on.

In our solution, we choose three open-source data stores, HBase, FastDFS and HDFS, which have different strengths for storing unstructured data, and can complement each other. Firstly, HBase is very suitable for storing Features of the unstructured data, because it is a column-based storage for hosting of very large tables, and the columns of HBase do not need to be defined at schema time but can be conjured on the fly. Secondly, HDFS is designed and optimized for very large files and prioritizes high throughput over low latency of data access with streaming access pattern, while FastDFS is a light-weight distributed file system for high capacity and load balancing, which is designed to store sources of the websites whose services are based on files, such as photo sharing and video sharing sites. Therefore, HBase and FastDFS together can manage both large and small files for the unstructured data.

For more details, as we know, the pervasive Hadoop version (not Federation) exploits one daemon (*NameNode*) to manage the whole namespace. To locate a file, the NameNodes name system lookups the mapping of file name to list of *DataNodes*. Different from a single name system of HDFS, FastDFS surrogates the file's location service to its users by a kind of token named *fileId*. The fileId is generated when a user uploaded a file, and the user needs to maintain it. It encodes the location of a group name and the file location in the group (and timestamp for file synchronization). The *Tracker Server*, acting as a master role, is responsible of decoding the location from the fileId when the user wants to download the file. By leveraging the fileId, more than one tracker server can be started in one FastDFS cluster. In our hybrid system, we maintain those fileIds (in FastDFS) or file names (in HDFS) of these unstructured data in HBase.

In order to prove the complementarity of these three stores, we evaluate the upload (write) and download (read) performance of the data stores with respect to the blob size. Given a data store and massive files with pre-specified sizes, we upload the file and then download them in parallel in multiple servers where the volume of data is 100GB. In order to eliminate differences caused by the configuration of physical machines, we deploy the key components of the three data stores on the same set of machines. When the upload or download phase finishes, the throughput of each server is computed and the average result is shown in Fig. 3.

According to Fig. 3(a), HBase has best write performance when the file is very small, while HDFS obtains best throughput when the file size get larger. The reason is that the key-value store HBase does well in storing small key-value pairs by its MemStore, while HDFS exploits block-oriented and pipelined writes to achieve high throughput. The throughput of FastDFS outperforms the other two as long as the blob size is between 10KB and 100KB. The read performance in Fig. 3(b) has the similar results. That is to say, FastDFS would achieve the best performance when blob size is between 5KB and 10MB. In addition, we also annotate with the minimum and maximum performance gains which indicate how the most suitable data store outperforms the others. For example, the throughput of FastDFS is 2.45 $\times$ better than that of HDFS and 3.38 $\times$ better than that of HBase when the blob size is 100KB.



(a) Write Performance (b) Read Performance

Fig. 3. The performance of three data stores.

2.4. Data placement

According to the above analysis, we find that various data stores have quite different strengths, so we should find out a suitable data placement policy.

Users can specify data stores via user-defined FeatureTypes in our hybrid storage architecture, but they may place the data in unsuitable stores. Here, we focus on the data placement policy to enhance the comprehensive performance of integrated data stores. Especially, since the size of the unstructured data can influence the performance of the storage system greatly, we used the size-based scheme for data placement in the hybrid storage architecture by default.

In our implementation, we defined two abstract interfaces, namely *StoreAccess* and *BlobStoreAccess* to manage structured data such as metadata, low-level features and high-level semantic features, and binary files. The former is responsible for the majority of value types except the blob type, which is handled by the latter. Now, we implemented HDFSBlobStoreAccess, FastDFSBlobStoreAccess, and HBaseBlobStoreAccess for managing the blob data of UObject, and HBaseStoreAccess for managing the structured data of UObject. In fact, we also implemented MySQLStoreAcess, but we found it is not suitable for the UObject management, since it does not support flexible data model well and it is not a column-based storage which has efficient batch processing capacity. For blob data management, we used size-based scheme by default. That is to say, since the performance of read operations would influence the hybrid storage system more in the write-once-read-many scenario, so when the size of the blob is between 5KB and 10MB, we used FastDFSBlobStoreAccess to manage these blobs according to the experimental results in Fig. 3. While if the blob is larger than 10MB, it will be stored in HDFS, and if the blob is smaller than 10KB, it will be stored in HBase.

Other data stores can also be integrated into the hybrid storage architecture by implementing the interfaces. Here, users may have different considerations in term of data placement, and we prioritize the user-specified scheme over the size-based scheme.

3. Efficient unstructured data processing

The hybrid storage architecture utilizes the characteristics of different underlying data stores and provides a unified data access interface. Based on it, this section presents an overview of a parallel data processing framework at first, and then describes several partitioning strategies which are used for splitting inputs of MapReduce job in an efficient manner.

3.1. Hadoop MapReduce framework background

Our parallel processing architecture for the hybrid storage is based on the Hadoop MapReduce framework.

The MapReduce framework follows a simple master-slave architecture. It consists of a single Master node that runs a JobTracker instance and several Slave nodes each running a TaskTracker instance. The JobTracker handles the MapReduce jobs by splitting each job into Map tasks and Reduce tasks, and assigning each task to a TaskTracker based on its load and available resources. So the tasks can be executed in parallel in the TaskTrackers.

The InputFormat library in Hadoop describes the input specification for a MapReduce job and defines how to read data from a file (or connect to arbitrary data sources) into the Mapper instances. For example, the default TextInputFormat reads lines of text files and transforms them into key-value pairs that Map tasks can process. In addition, Users can develop a custom InputFormat that reads files of a particular format or files from arbitrary sources.

3.2. Parallel processing architecture

Our parallel processing architecture (see Fig. 4) integrates seamlessly with Hadoop's MapReduce framework by extending Hadoop's InputFormat and using storage manager in our unified storage framework.

Upon receiving users' analysis or index tasks on a set of UObjects, the hybrid storage and processing layer (HSPL) master prepares configurations, such as fetching meta data and statistics from the *Metastore*, and submits a job to MapReduce

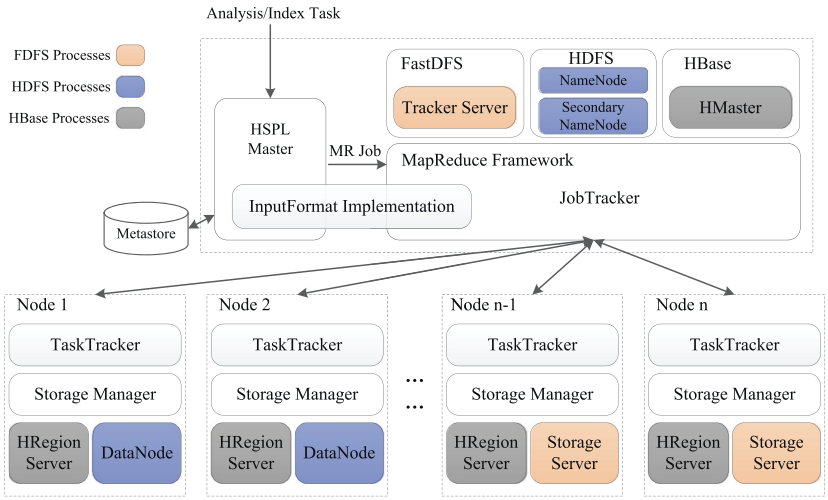


Fig. 4. Overview of parallel data processing framework.

framework. The JobTracker employs the InputFormat to generate splits of the data. A split is the unit of scheduling and is a non-overlapping partition of the input data. Each split is then assigned to a map task to be processed.

Fig. 4 depicts two layers in each physical node: (1) a data storage layer and (2) a data processing layer. Each node is configured to be TaskTracker, whereas the data may reside in different kinds of data stores. Indeed, the TaskTrackers need not to know whether the data is in HDFS or FastDFS. Based on the unified data storage framework, the TaskTrackers use storage manager to access data in a manner that is similar to fetching data blocks in HDFS in Hadoop framework. After that, the data is organized in the form of UObject and processed according to users' task requirements.

In this architecture, partitioning strategies can be implemented by developing customized InputFormats, since the InputFormat is responsible for creating the input splits and dividing them into records for MapReduce tasks in the Hadoop MapReduce framework. In the following subsections, we will discuss three different InputFormat implementations for partitioning strategies in the hybrid storage system.

3.3. Partitioning strategies for parallel processing

As to a table in OLTP databases, the most common strategy to horizontally partition a table is hashing or range partitioning. It usually works well in many distributed databases by hashing on primary key or partitioning numeric features. We stored blob feature and other primitive features in a logical table in HBase, however, it makes no sense to use hashing or range partitioning directly on blob feature because its value is encoded location instead of numeric data types.

TableInputFormat was implemented on an HBase table to execute a MapReduce job. However, we could not use it directly to process unstructured data since the region-based granularity is too coarse. That is to say, it assigns each region of a table to one map task. Hence, if there is only one region in the table, such as in video feature extraction scenario where the amount of location information is small but the amount of videos is very large, there would be only one map task for the job.

In the following section, we discussed three partition strategies to address the above problem, including *Key Range based Partitioning Strategy, Group based Workload Balanced Partitioning Strategy* and *Group based Execution Time Balanced Partitioning Strategy*.

3.3.1. Key range based partitioning strategy

In order to control the number of splits, we implemented a custom InputFormat (*KeyRangeInputFormat*), which uses a key range based partitioning strategy. It splits each region of a table into multiple segments and each segment is represented by a key range. Thus, each partition consists of a list of key ranges. Note that if one region is too small to be split into the specified number of partitions, it will be assigned to one partition. Fig. 5 gives an example of splitting two regions into three partitions.

In the figure, UObjects spread over two regions in HBase, and their blobs are stored in the three types of stores including HBase, FastDFS and HDFS respectively, which are denoted by shapes such as triangle, pentagram and hexagon. Then, each partition is composed of several segments, which are all represented by a key range. For example, *Partition 1* in the Fig. 5 is composed of a set of UObjects, whose row key ranges are [r1001–r1300] and [r2001–r2400]. Both *Partition 1* and *Partition 2* have the same number of UObjects, but the corresponding blobs are stored in different stores. In this circumstance, when reading data in one partition, the RecordReader needs to access different data stores and pays for context switches. In

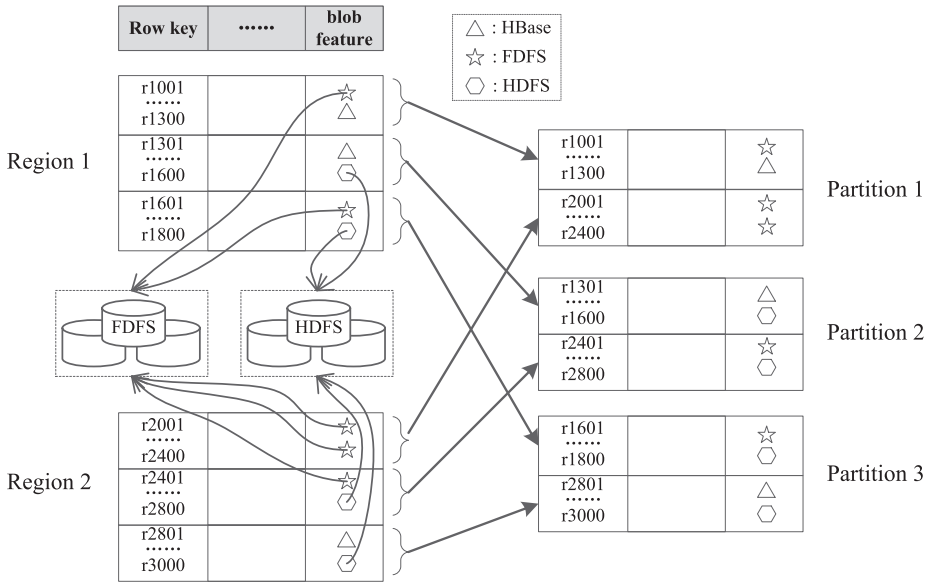


Fig. 5. Key range based partitioning strategy.

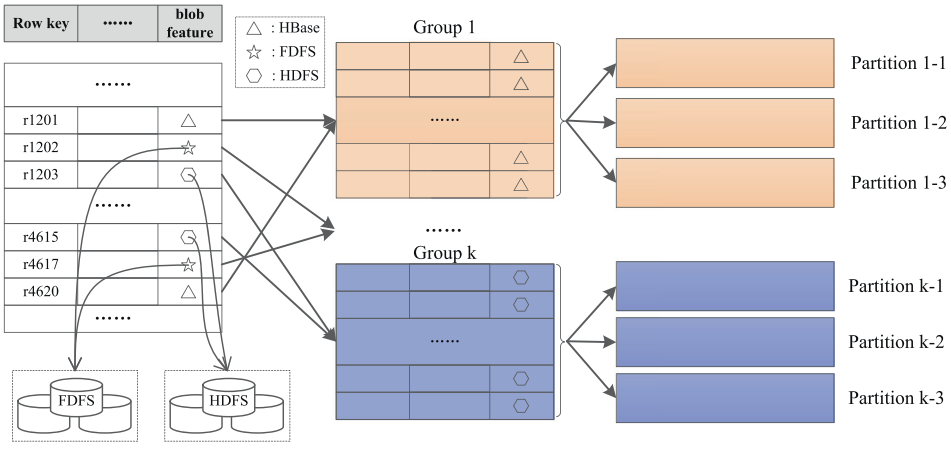


Fig. 6. Group based workload balanced partitioning strategy.

addition, key range based partitioning cannot ensure that each key range has equal number of keys. Even though the number of keys is equal, the distribution of data size of unstructured data in one partition is always skewed.

Therefore, key range based partitioning is a generalized strategy for data in a logical table, but it is not optimal in the hybrid storage architecture.

3.3.2. Group based workload balanced partitioning strategy

In order to address the problems in the *Key Range based Partitioning Strategy*, we proposed the *Group based Workload Balanced Partitioning Strategy*, which considers the characteristics of different data stores.

In the strategy, we assume that a collection G contains the keys of all blobs required to be processed, and they are stored in K kinds of data stores. Each data store manages a subset G_i of keys, where $\cup_{i=1}^{K} G_i = G$ and $G_i \bigcap G_j = \phi$ (where $i \neq j$). Before the batch data processing, all keys are split into M partitions. The first step needs to know the blobs in G_i maintained in each data store and the total volume s_i of data in a specific table. This information is obtained from the statistic table. The second step computes the number of partitions m_i for each data store, where $m_i = \frac{s_i}{\sum_{j=1}^{K} s_j} * M$. The third step is to group the keys into a partition. In order to ensure each partition is workload balanced, statistic information in each data store is sorted by blob size and split into partitions evenly, the volume of which equals $\frac{s_i}{m_i}$ $(i = 1 \ldots K)$. Fig. 6 shows an example of this partitioning strategy.

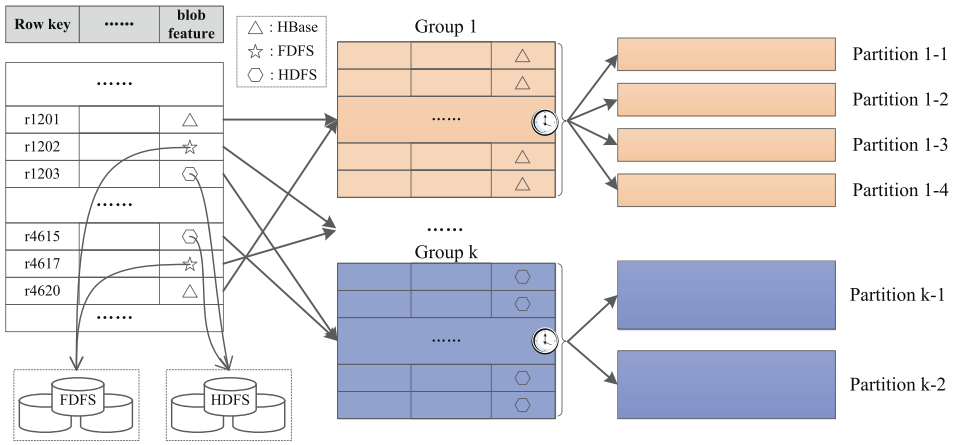


Fig. 7. Group based execution time balanced partitioning strategy.

Comparing to the Fig. 5, we found UObjects having blob in the same data store are grouped at first as shown in Fig. 6. For example, UObjects having blob stored in HBase are grouped into *Group 1*, while UObjects having blob stored in HDFS are grouped in *Group k*. Then, partitions are generated from the groups by considering the blob size.

3.3.3. Group based execution time balanced partitioning strategy

In addition to the volume of data, the performance difference in terms of processing unstructured data in different stores should also be considered for data partitioning. This is expected to be more reasonable since our goal is to balance the execution time of each map task. Thus, we proposed *Group based Execution Time Balanced Partitioning Strategy* by considering both the volume of data and the execution time.

Compared to the three steps in *Group based Workload Balanced Partitioning Strategy*, this partitioning strategy only differs in the second step. It computes the number of the partitions for each data store by additionally considering the fetch time for the unstructured data, since the performance is quite different for different stores as discussed in Section 2.4.

According to the amount of data and reading performance of each data store, this partitioning strategy estimates the total execution time, which is formulated as $\frac{s_i}{ps_i}$ $(i = 1 \ldots K)$ and ps_i can be viewed as the processing speed of the ith data store. Here, ps_i can be estimated by the read performance as in Fig. 3(b). Thus, the number of partitions m_i for the ith data store is computed using $m_i = \frac{s_i/ps_i}{\sum_{j=1}^{K}(s_j/ps_j)} * M$. Fig. 7 depicts the procedure of this strategy, where the first step is also the UObject grouping according to the type of data store as in Fig. 6, but the number of partitions from one data store depends on the total execution time.

4. Experimental evaluation

We evaluated the performance of our hybrid storage solution by comparing it with a common Hadoop solution. Following that, we investigated three partitioning strategies and studied their differences by a batch processing job on a common Hadoop solution and our hybrid solution.

4.1. Experimental environment

Cluster setup. Fig. 8 describes the logical view of our whole cluster setup. It contains clients, an ensemble of service daemons, and three ensembles of servers of data storage and processing.

However, some ensembles of servers were physically shared on a cluster of ten commodity machines. In the experiment of performance evaluation, clients and storage managers were responsible for submitting requests and routing them. These two ensembles of servers were distributed on the whole cluster. The three ensembles of data storage servers were shared, where one machine acted as the master and other nine machines were slaves. Besides, one JobTracker and ten TaskTrackers were used for data processing. All machines in the cluster were Inter Xeon CPU (2.80 GHz, 8 cores) with 48GB memory and 2TB SATA disks, and interconnected with 1Gbps Ethernet network.

For FastDFS, we studied the impact of different number of tracker servers by uploading and downloading a set of files and measuring the throughput. The results in Fig. 9 demonstrate that the write throughput is almost stable as the number of tracker servers is increased from one to three, while the read performance improves by about 20% as the number increases. So in the experiments, FastDFS was configured with three tracker servers and three groups, where each group had three storage servers.

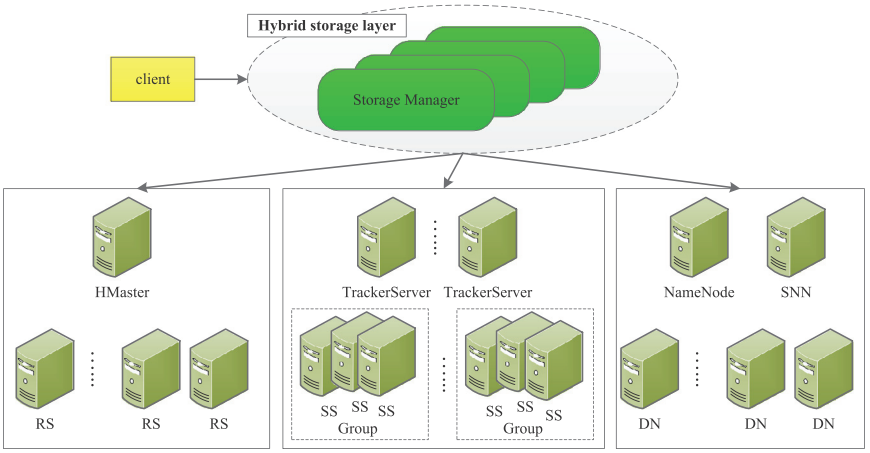


Fig. 8. Logical view of cluster setup of the hybrid storage architecture.

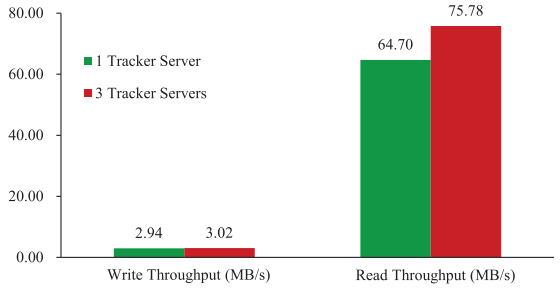


Fig. 9. Impact of different number of tracker servers.

Table 1
Distribution of three datasets.

	Sougou Pictures		Youtube Videos		CADAL Arts	
	Num	Size(KB)	Num	Size(KB)	Num	Size(KB)
< 1KB	1069	520	0	0	0	0
1KB → 10KB	119,376	758,923	0	0	9	72
10KB → 100KB	1,565,548	67,148,765	13	954	21	867
100KB → 1MB	311,713	61,224,375	622	405,401	1540	1,158,396
1MB → 10MB	2294	3,127,708	15,852	90,909,917	37,320	252,811,629
10MB → 50MB	0	0	24,494	569,556,391	4297	112,207,722
50MB → 100MB	0	0	4139	289,935,552	1803	131,260,591
> 100MB	0	0	3186	795,633,493	10	2,877,422
Total number	**2,000,000**		**48,306**		**45,000**	
Total data size	**132,260,291**(KB)		**1,746,441,708**(KB)		**500,316,699**(KB)	

Software configuration. The cluster ran 64-bit Linux. We used HBase with version 0.94.1, Zookeeper with version 3.4.3, Hadoop with version 1.0.3 and FastDFS with version 4.05. The block size of HDFS was 64MB as default. Hadoop was configured to run six map slots (mappers) and six reduce slots (reducers) per node. In addition, we turned off speculative execution for MapReduce jobs.

Datasets. We used three different datasets in the following experiments. The first one is Sougou picture dataset[3] which contains 20 million pictures from the internet and the dataset size is 650GB. The second is video dataset which contains 1665GB videos crawled from Youtube.com. The third dataset is a partial set of Chinese arts likes Chinese calligraphy works and paintings from CADAL.[4] We picked 45,000 pieces of work with compressed file format and the dataset size was 500GB.

Table 1 describes the fine-grained distribution of three datasets, which is similar to the statistic information based on a certain data store. The cumulative distribution of data size is shown in Fig. 10. The statistics here can be used for helping us analyze the results of different partitioning strategies. We find that the size of 98% of Sougou pictures is less than 1MB.

[3] http://www.sogou.com/labs/resources.html.
[4] http://www.cadal.zju.edu.cn.

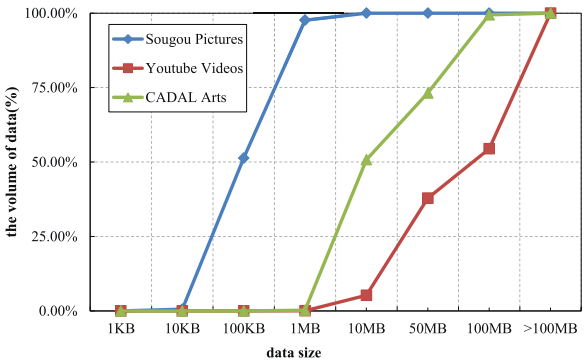


Fig. 10. Cumulative distribution function of data size.

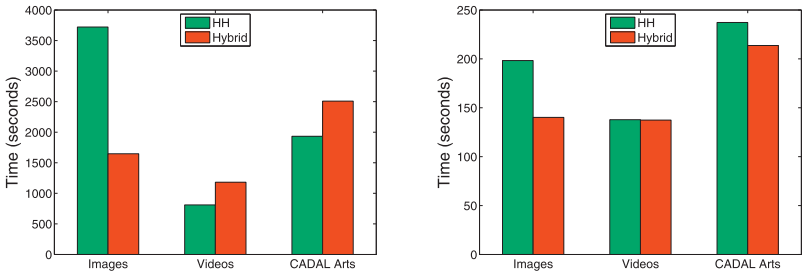


Fig. 11. Elapsed time on three datasets.

It makes FastDFS be a primary data store. By contrast, Youtube videos and CADAL arts having many large files (more than 10MB) result in that both FastDFS and HDFS are used in the underlying storage system. The distribution also shows that it is easier on Sougou pictures dataset than on CADAL arts dataset to get balanced partitions. It can been observed in the following Section 4.3.

4.2. Evaluating the hybrid storage

As illustrated in Section 2.4, it identifies a performance gap between different data stores in terms of reading and writing data. In this section, we evaluated the performance of the hybrid storage solution by comparing it with a common Hadoop solution (HH), where structured data and unstructured data of small size is stored in HBase in the form of bytes and data of large size is stored in HDFS as a separate file.

Fig. 11 depicts the elapsed time that a single client uploads three different types of data and downloads them, where each type of data is selected randomly from the three datasets and the volume of each category is 3GB (50,000 images), 15GB and 15GB respectively.

Note that there is a trade-off between fast data uploading and data downloading due to a configuration of the size-based scheme. Since the goal is to reduce overhead of reading unstructured data many times, we trade data downloading for data uploading. The results show that the hybrid solution saves up to 125% of uploading time and 41% of downloading time on the Image dataset. The reason is that the size of images is distributed mainly between 10KB and 1MB where the common Hadoop solution does not provide best storage performance. On the other two datasets, the hybrid solution has poor uploading performance yet a competitive or better downloading performance.

In fact, the downloading performance gain depends on several factors, including the order of uploading data, the distribution of the uploaded data and whether the characteristics of data stores are considered. Following this, we focus on data distribution at first and then consider the characteristics of data stores in Section 4.3. According to statistics, the proportion of files, the size of which is between 5KB and 20MB, is about 15% in Youtube Video dataset and 50% in CADAL Arts dataset. Since we also exploit HBase and HDFS in the hybrid solution, the performance gain of FastDFS is amortized. Therefore, we increase the proportion while fixing the total volume of data and plot the elapsed data downloading time in Fig. 12.

The results show when the proportion reaches 80%, the hybrid solution saves up to 18% of downloading time on Video dataset and 11% on Arts dataset. As expected, the bigger the proportion, the more the performance gain of the hybrid solution.

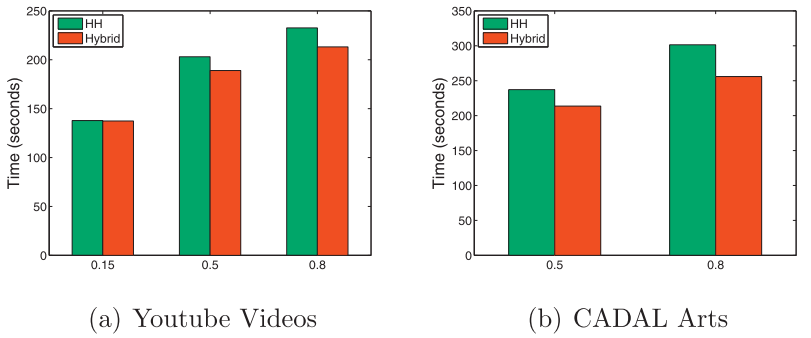


Fig. 12. Data downloading time in terms of proportions of files (the size is between 5KB and 20MB) on two datasets.

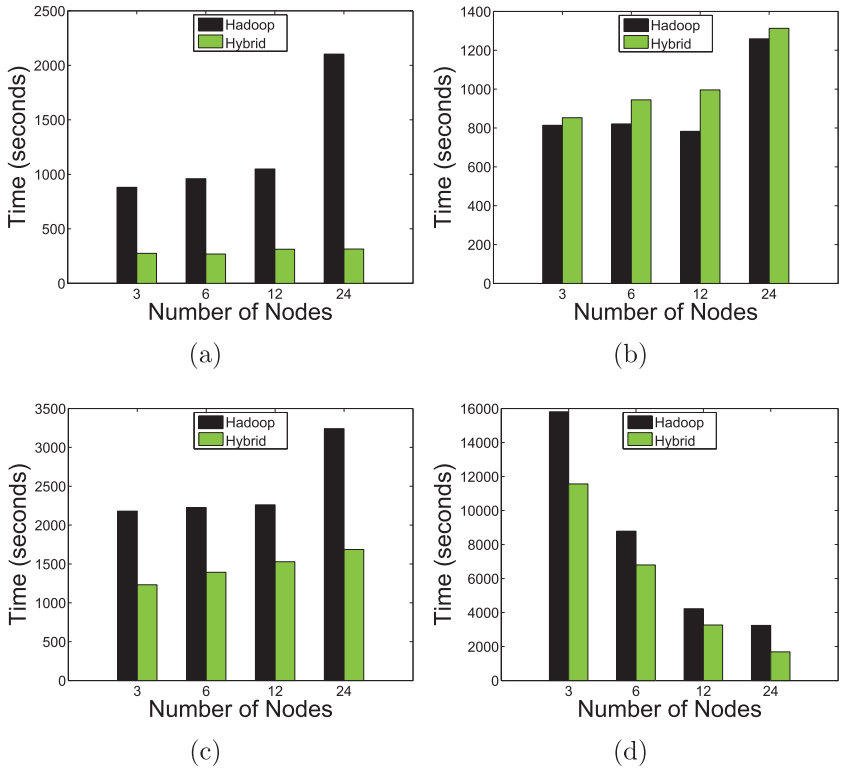
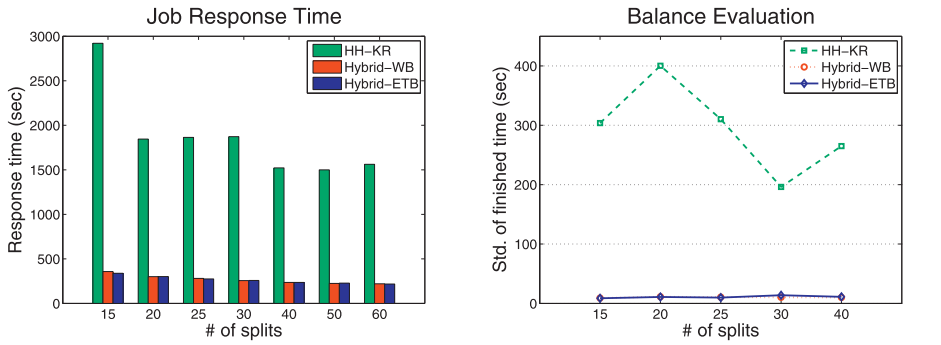


Fig. 13. Load times: (a) Image Set (500MB/node), (b) Video Set (10GB/node), (c) Mixed Set (500MB images + 10GB videos, per node), and (d) Mixed Set (252GB/cluster).

In addition, we also evaluated the hybrid storage system with different cluster size. We adopt two approaches for data loading. The first one fixes the size of data per node to be the same and only varies the number of nodes from 3, 6, 12 to 24. The second fixes the total dataset size to be the same and evenly divides the data amongst a variable number of nodes. The results are shown in Fig. 13.

The results for loading images and videos are shown in Fig. 13(a) and (b), where the size of data per node is fixed to be 500MB and 10GB, respectively. We can observe that for small files like images, our hybrid storage system outperforms Hadoop by a factor of 3.2 × to 6.7 × since the hybrid storage system employs suitable data stores. However, as shown in Fig. 13(b), our hybrid system takes more time to load large video files than Hadoop. This is because our hybrid system still store the video files into the HDFS but having several additional operations such as storage mapping. Nevertheless, the overhead is about 13%, which is acceptable.

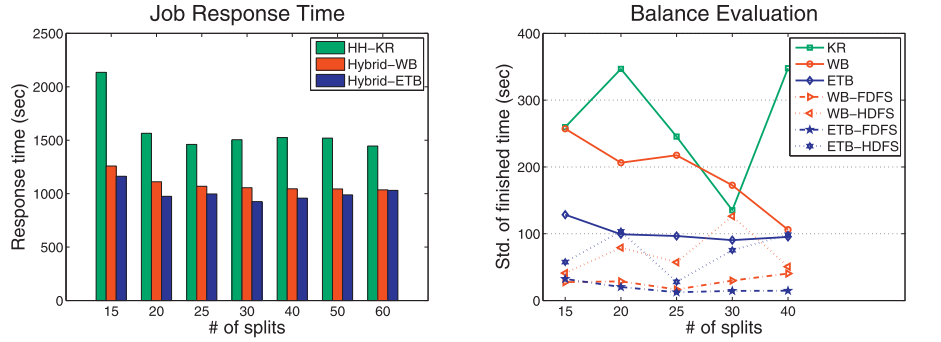
In addition, we also load different kinds of data simultaneously in each node, and the results are depicted in Fig. 13(b). It shows that our hybrid system also outperforms Hadoop up to 1.9 ×, which attributes to the appropriate storage decisions of the hybrid storage system. Meanwhile, We also evaluated the scalability of two systems by fixing the total dataset size at 252GB and dividing the data evenly amongst nodes. Fig. 13(d) shows the results on the mixed data set. It can be seen



(a) Job response time

(b) Standard deviation of finished time of mappers

Fig. 14. Job execution on Sougou pictures.



(a) Job response time

(b) Standard deviation of finished time of mappers

Fig. 15. Job execution on CADAL Arts.

that as the number of nodes increases, the loading times of our hybrid storage system and Hadoop both decrease, which demonstrates the scalability of our hybrid storage system.

4.3. Comparing the partitioning strategies

We studied the performance of three partitioning strategies by measuring the response time of running a dummy MapReduce job with different numbers of partitions or *InputSplits*. The dummy MapReduce job scans through the input dataset without further processing. That is to say, the dummy job only considers the data loading procedure from different stores but ignores the data processing procedure, since our paper mainly focuses on the hybrid storage system for different unstructured data. Thus, we evaluated the influence of the hybrid storage system in the MapReduce framework with the dummy jobs by only considering the data loading time.

In the experiments, we compared the performance of our hybrid storage system with a common Hadoop solution using the three partitioning strategies: (1) a Hadoop solution including HBase and HDFS executes the dummy job by key range based strategy (HH-KR), (2) our hybrid storage system exploits the group based workload balanced partitioning strategy (Hybrid-WB), and (3) the hybrid solution uses the group based execution time balanced strategy (Hybrid-ETB). Both the Hadoop and hybrid solution choose the most suitable data store to place the data by the results of above experiment. The experimental results on Image Set and CADAL Art Set are shown in Figs. 14 and 15.

Fig. 14(a) shows the dummy job's response time for 2,000,000 Sougou pictures. Due to the benefits of parallelism, the needed time decreases as the number of splits increases for all three partitioning strategies. The two balanced partitioning strategies have almost the same results since the difference of performance of data stores has not yet changed the distribution of partitions. However, both of them outperform the Hadoop solution with key range partitioning by a factor of 6.1 × to 8.6 ×. It attributes to the efficiency of the hybrid storage architecture as well as the balanced partitioning strategy. The balance evaluation is shown in Fig. 14(b) when the number of splits is from 15 to 40. We do not compute the standard deviation if the number of partitions in one data store is less than two. We observe that HH-KR has higher variation than the others, which results in long response time by some slow mappers.

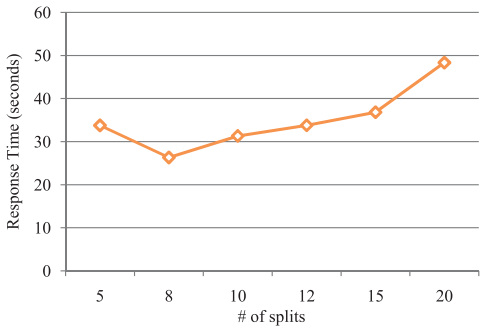


Fig. 16. Average processing time of CEDD feature extraction jobs with Hybrid-ETB.

A slightly different result of job's response time for the CADAL Art Set is shown in Fig. 15(a). As the degree of parallelism increases, the needed time reduces slightly because reading large blob files by InputStream occupies the disk IO and the bandwidth. Both the Hadoop solution and the hybrid solution place the large blobs in HDFS, but the Hybrid-ETB's performance still improves over HH-KR by at most 84%. Besides, the execution time balanced strategy outperforms the workload balanced strategy by around 10% because of the difference of FastDFS and HDFS. Fig. 15(b) demonstrates the standard deviation of mappers' finished time. It shows that the deviation of key range based partitioning is still the highest and that of ETB is the lowest.

Considering the standard deviation separately in different data stores, the finished time of mappers in FastDFS is more balanced than that in HDFS. The reason is that splitting keys of small blobs evenly into partitions is easier than that of large ones.

In addition, we also evaluated the hybrid storage system with a real MapReduce job, image CEDD (Color and Edge Directivity Descriptor) feature extraction. In the experiment, we executed the MapReduce jobs 6 times, and then calculated the average processing time to reduce the noise caused by the experiment environment such as network jitter. Fig. 16 shows the average processing time of CEDD feature extraction for 100,000 images with different number of map splits.

From the figure, we can find it takes about 35 s to extract CEDD features for 100,000 images, but it costs about 18.17 min when we used HH-KR solution. In addition, we found when the number of splits equals to 8, it reaches the best performance. This is because fewer splits will lead to fewer map jobs and slow the processing, but more jobs will bring more JVM startup overhead.

5. Related work

In this section, we provide an overview of related work in the field of hybrid storage management and data processing optimization.

Many approaches are proposed to build a hybrid system, which may integrate file systems, traditional databases, NoSQL stores and cloud storage. Alekh et al. [6] launch a project named OctopusDB, the goal of which is to free users from the details of data storage through a virtual storage layer [7]. The virtual storage layer provides a unified storage framework for several use-cases including file system, RAID, RDBMS and cloud storage. As they indicated, the most critical component, a data storage optimizer, takes care of all data storage decisions for what, where and how to store data. It depends on the hints from users in the form of storage themes. Thus, a new declarative data storage language is required and designed to deliver the user hints. Polybase [8] is designed to enable the data processing in SQL Server PDW. It unifies non-relational data in the Hadoop cluster and traditional relational data in RDBMS. The basic idea of data processing behind Polybase is to choose MapReduce-style schemes or parallel databases, based on the cost of data movement and query execution. HadoopDB [9] is a special hybrid of parallel DBMS and Hadoop. It connects multiple single-node DBMS using Hadoop for inheriting MapReduce-style scalability and fault-tolerance, as well as the performance of DBMS. HadoopDB exploits MapReduce as the task coordinator and network communication layer. Queries are parallelized across the nodes via MapReduce. The goal of HadoopDB is data analysis, and the data is actually stored in the DBMS nodes. Data processing is boosted by the features of single-node DBMS engines. BigDAWG Polystore System [10] was presented as a new view of federated databases to address the growing need for managing information that spans multiple data models. It can offer full functionality and location transparency through the use of explicit scopes and casts. Odyssey [11] and MISO [12] also employ a multi-store approach to store and query data, but they focus on the query processing on multi-stores and do not focus on the management of the different types of unstructured data. In addition, Octopus [13] was proposed as a hybrid data processing engine to fully integrate backend systems for query processing. The key of Octopus is to optimize the amount of data movement across backend systems.

Some research has focused on building a hybrid system selectively using storage devices with different performance characteristics [14–16]. Tian et al. [14] implement a heterogeneity-aware framework for DBMS storage management called hStorageDB, which is organized into a two-level hierarchy consisting of SSDs and HDDs. hStorageDB extracts semantic in-

formation from the query optimizer and query planner and passes it with IO requests to the storage manager. It leverages semantic information to make effective data placement decisions in storage systems. Xin et al. [15] introduce hybrid storage management for database systems in which the two types of devices (SSD and HDD) are visible to the database management system. Therefore, it can use the information at its disposal to decide how to make use of the two types of devices. Recently, HDFS community is working on an issue[5] which aims at adding support for heterogeneous storage types, such as HDD, RAM, RAID, remote storage (NAS) etc.

Other studies on the performance of storage systems are concerned with their physical data organization. They often utilize hybrid data layout schemes for efficient storage management, which can be divided into two categories. One is memory-based storage systems [17–20]. For example, SAP HANA database [21,22] has one in-memory storage engine, Relational Engine, which supports both row-oriented and column-oriented physical representations. The other is disk-based storage systems [23–26]. For example, Fractured Mirrors approach [23] maintains two copies of the database-one is column-oriented and one row-oriented copy. Based on the query characteristics, it determines where to execute the query. These two categories both utilize database-like data processing without any aspect of MapReduce jobs.

In addition, due to the use of UDFs in databases, many databases have more functions than before. For example, Madlib [27] tries to provide a suite of SQL-based algorithms for machine learning, data mining and statistics for structured and unstructured data. However, Madlib does not concern where to efficiently store different types of data and how to partition data for fast data processing.

6. Conclusions

In this paper, we present a hybrid storage architecture, which integrates various kinds of data stores to model, store and process the unstructured data. Based on this unified hybrid storage system, several partitioning strategies are proposed to execute MapReduce-based batch-processing jobs. We demonstrate that it is possible to build a smart and efficient system by utilizing the characteristics of different data stores. By leveraging the unified storage system, the partitioning strategies with statistic information of unstructured data placement reduce the IO cost and substantially improve the efficiency of large-scale data processing.

Acknowledgement

The work in this paper was supported by the National Science and Technology Major Project of China under Grant No.2010ZX01042-002-003-001, the Zhejiang Provincial Natural Science Foundation of China (No. LY17F020015), the Fundamental Research Funds for the Central Universities (No. 2017FZA5016), and the Chinese Knowledge Center of Engineering Science and Technology (CKCEST).

References

[1] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google file system, in: ACM SIGOPS Operating Systems Review, 37, ACM, 2003, pp. 29–43.
[2] Apache Hadoop, (http://hadoop.apache.org/).
[3] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[4] Apache hbase, (http://hadoop.apache.org/hbase/).
[5] Q. Yu, Fastdfs: framework analysis and configuration optimization, 2012, Dtcc.
[6] Octopusdb project, (https://infosys.uni-saarland.de/projects/octopusdb.php).
[7] A. Jindal, J.-A. Quiané-Ruiz, J. Dittrich, Wwhow! freeing data storage from cages, CIDR, 2013.
[8] D.J. DeWitt, A. Halverson, R. Nehme, S. Shankar, J. Aguilar-Saborit, A. Avanes, M. Flasza, J. Gramling, Split query processing in polybase, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 1255–1266.
[9] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, Hadoopdb: an architectural hybrid of MapReduce and dbms technologies for analytical workloads, Proc. VLDB Endow. 2 (1) (2009) 922–933.
[10] J. Duggan, A.J. Elmore, M. Stonebraker, M. Balazinska, B. Howe, J. Kepner, S. Madden, D. Maier, T. Mattson, S. Zdonik, The bigdawg polystore system, ACM SIGMOD Rec. 44 (2) (2015) 11–16.
[11] H. Hacígümüş, J. Sankaranarayanan, J. Tatemura, J. LeFevre, N. Polyzotis, Odyssey: a multistore system for evolutionary analytics, Proc. VLDB Endow. 6 (11) (2013) 1180–1181.
[12] J. LeFevre, J. Sankaranarayanan, H. Hacigumus, J. Tatemura, N. Polyzotis, M.J. Carey, Miso: souping up big data query processing with a multistore system, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, ACM, 2014, pp. 1591–1602.
[13] Y. Chen, C. Xu, W. Rao, H. Min, G. Su, Octopus: hybrid big data integration engine, in: 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2015, pp. 462–466.
[14] T. Luo, R. Lee, M. Mesnier, F. Chen, X. Zhang, hstorage-db: heterogeneity-aware data management to exploit the full capability of hybrid storage systems, Proc. VLDB Endow. 5 (10) (2012) 1076–1087.
[15] X. Liu, K. Salem, Hybrid storage management for database systems, Proc. VLDB Endow. 6 (8) (2013) 541–552.
[16] A. Badam, V.S. Pai, Ssdalloc: hybrid ssd/ram memory management made easy, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, USENIX Association, 2011. 16–16
[17] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, S. Madden, Hyrise: a main memory hybrid storage engine, Proc. VLDB Endow. 4 (2) (2010) 105–116.
[18] M. Zukowski, N. Nes, P. Boncz, Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing, in: Proceedings of the 4th International Workshop on Data Management on New Hardware, ACM, 2008, pp. 47–54.
[19] A. Ailamaki, D.J. DeWitt, M.D. Hill, M. Skounakis, Weaving relations for cache performance, in: VLDB, 1, 2001, pp. 169–180.

[5] https://issues.apache.org/jira/browse/HDFS-2832.

[20] R.A. Hankins, J.M. Patel, Data morphing: an adaptive, cache-conscious storage technique, in: Proceedings of the 29th International Conference on Very Large Data Bases-Volume 29, VLDB Endowment, 2003, pp. 417–428.
[21] F. Färber, S.K. Cha, J. Primsch, C. Bornhövd, S. Sigg, W. Lehner, Sap hana database: data management for modern business applications, ACM SIGMOD Rec. 40 (4) (2012) 45–51.
[22] P. Rösch, L. Dannecker, F. Färber, G. Hackenbroich, A storage advisor for hybrid-store databases, Proc. VLDB Endow. 5 (12) (2012) 1748–1758.
[23] R. Ramamurthy, D.J. DeWitt, Q. Su, A case for fractured mirrors, VLDB J. Int. J. Very Large Data Bases 12 (2) (2003) 89–101.
[24] A. Jindal, J.-A. Quiané-Ruiz, J. Dittrich, Trojan data layouts: right shoes for a running elephant, in: Proceedings of the 2nd ACM Symposium on Cloud Computing, ACM, 2011, p. 21.
[25] E. Thereska, P. Gosset, R. Harper, Multi-structured redundancy, in: Proceedings of the 4th USENIX Conference on Hot Topics in Storage and File Systems, HotStorage'12, USENIX Association, Berkeley, CA, USA, 2012. 1–1
[26] J. Schaffner, A. Bog, J. Krüger, A. Zeier, A hybrid row-column oltp database architecture for operational reporting, in: Business Intelligence for the Real-Time Enterprise, Springer, 2009, pp. 61–74.
[27] J.M. Hellerstein, C. Ré, F. Schoppmann, D.Z. Wang, E. Fratkin, A. Gorajek, K.S. Ng, C. Welton, X. Feng, K. Li, A. Kumar, The madlib analytics library: or mad skills, the sql, Proc. VLDB Endow. 5 (12) (2012) 1700–1711, doi:10.14778/2367502.2367510.