Frontiers of Information Technology & Electronic Engineering
www.zju.edu.cn/jzus; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

Pegasus: a distributed and load-balancing fingerprint identification system*#

Yun-xiang ZHAO†, Wan-xin ZHANG†, Dong-sheng LI†‡, Zhen HUANG†, Min-ne LI†, Xi-cheng LU

(*National Laboratory for Parallel and Distributed Processing, College of Computer, National University of Defense Technology, Changsha 410003, China*)

†E-mail: zhaoyx1993@163.com; camu7s@163.com; dsli@nudt.edu.cn; maths_www@163.com; litoeknee@gmail.com

Received Dec. 29, 2015; Revision accepted Apr. 12, 2016; Crosschecked July 11, 2016

Abstract: Fingerprint has been widely used in a variety of biometric identification systems in the past several years due to its uniqueness and immutability. With the rapid development of fingerprint identification techniques, many fingerprint identification systems are in urgent need to deal with large-scale fingerprint storage and high concurrent recognition queries, which bring huge challenges to the system. In this circumstance, we design and implement a distributed and load-balancing fingerprint identification system named Pegasus, which includes a distributed feature extraction subsystem and a distributed feature storage subsystem. The feature extraction procedure combines the Hadoop Image Processing Interface (HIPI) library to enhance its overall processing speed; the feature storage subsystem optimizes MongoDB's default load balance strategy to improve the efficiency and robustness of Pegasus. Experiments and simulations are carried out, and results show that Pegasus can reduce the time cost by 70% during the feature extraction procedure. Pegasus also balances the difference of access load among front-end mongos nodes to less than 5%. Additionally, Pegasus reduces over 40% of data migration among back-end data shards to obtain a more reasonable data distribution based on the operation load (insertion, deletion, update, and query) of each shard.

Key words: Distributed fingerprint identification, Distributed MongoDB, Load balancing
http://dx.doi.org/10.1631/FITEE.1500487 **CLC number:** TP316.4

1 Introduction

Fingerprint identification is one of the most widely used methods in the field of biometric authentication thanks to its uniqueness, stability, and easy-to-collect (Shu *et al.*, 2014). It obtains fingerprints through hardware devices with optical sensor technology and then extracts the corresponding features like minutiae for the future matching phase (Hong *et al.*, 1998).

Existing fingerprint identification systems and related research have focused on parallelizing the process of preprocessing or feature extraction (Indrawan *et al.*, 2011), which increases single fingerprint's processing speed substantially. However, due to the scenarios of massive users and concurrent access with the popularity of fingerprint in biometric identification systems and the coming of big data age, it is essential to take distributed or cluster computing into consideration. Nowadays, public security systems intend to collect and store fingerprints of all people around a province to assist in identity recognition. In such applications, the number of users can rise up to over 10 millions with fingerprints of over

‡ Corresponding author

* Project supported by the National Basic Research Program (973) of China (No. 2014CB340303), the National Natural Science Foundation of China (Nos. 61222205 and 61402490), the Program for New Century Excellent Talents in University, China (No. 141066), and the Fok Ying-Tong Education Foundation

A preliminary version was presented at the 15th International Conference on Algorithms and Architectures for Parallel Processing, Zhangjiajie, China, Nov. 18–20, 2015

ORCID: Dong-sheng LI, http://orcid.org/0000-0001-9743-2034

100 millions, which makes it impossible to be handled by only one computer (Li *et al.*, 2015; Zhang *et al.*, 2016). Current research in distributed fingerprint identification systems is still at its beginnings for the following two main reasons:

First, it is costly to use multiple nodes for parallelizing the feature extraction phase among small but massive files (data transfer between nodes occupies a lot of time and resources in the whole procedure of distributed tasks). Some research distributed images into computing nodes to reduce the time consumption due to the advantage of distributed systems on multitask. However, transferring fingerprints (approximately 40 KB per fingerprint) between nodes is very time consuming.

Second, there should be an effective and load-balancing database to deal with large-scale concurrent data access at the server side. Existing databases consider only the volume of data among nodes to balance data storage without paying attention to the pressure of data accessing like query and insertion, which may cause breakdowns of nodes under high accessing pressure.

In view of these facts, we put forward a distributed and load-balancing fingerprint identification system named Pegasus. Pegasus contains two subsystems: a distributed feature extraction subsystem and a distributed feature storage subsystem (front-end mongos nodes and back-end data shards).

The distributed feature extraction subsystem uses the Hadoop Image Precessing Interface (HIPI) library to boost the processing speed. HIPI is an image processing library designed to be used with the Apache Hadoop MapReduce parallel programming framework (Sweeney *et al.*, 2011). It reduces small images' transfer time among Hadoop Distributed File System (HDFS) data nodes by compressing images into a Hipi Image Bundle (HIB).

The distributed feature storage subsystem constructs its database on a distributed MongoDB cluster. Distributed MongoDB consists of front-end mongos nodes (controlling the access requests from different users, such as query requests and insert requests), back-end data shards (storing the features of fingerprints), and config servers (storing meta data) (Plugge *et al.*, 2010). It is responsible for low time expense, robust storage and management of fingerprint features. The details of how Pegasus improves the load-balancing strategies for concurrent access requests (among front-end mongos nodes) and distributed feature data storage (among back-end data shards) will be discussed in Section 5.

The prophase work of Pegasus, named DFIS, has been presented in Zhao *et al.* (2015). Accordingly, Pegasus implements the aforementioned two subsystems and the reasons of its good performance will be expatiated theoretically in Sections 4 and 5. Additionally, Pegasus optimizes the data transmission between these two subsystems, which formats and then inserts the features of fingerprints into the distributed storage subsystem directly after the feature extraction procedure. Compared with DFIS, Pegasus realizes about 40% time reduction when transmitting data between the two subsystems.

In this study, we propose a distributed and load-balancing fingerprint identification system. The main contributions of Pegasus are as follows:

1. Implement the feature extraction of fingerprints with the HIPI library, which degrades the time consumption of processing massive fingerprints by about 70% compared with Hadoop.

2. Use consistent hashing to balance access requests on front-end mongos nodes, which makes the difference of access load less than 5%.

3. Optimize distributed MongoDB's default load-balancing algorithm from the view of data operation of each shard, which reduces over 40% data migration among data shards to gain a much more load-balancing state.

2 Related work

There exists much research in distributed image processing using distributed development platforms including Hadoop and Spark, such as the large-scale graphics processing based on Spark (Mader *et al.*, 2014), and the combination of Compute Unified Device Architecture (CUDA) and Hadoop for fast distributed image processing (Malakar and Vydyanathan, 2013). While these works do facilitate image processing in distributed systems, it is still challenging and worth studying to process small images including those of fingerprints in a high performance manner.

Currently, researchers in the fingerprint identification filed focus on reducing the feature

extraction time and raising the matching accuracy of traditional stand-alone fingerprint identification systems (Galar *et al.*, 2015b). For example, Indrawan *et al.* (2011) parallelized the process of minutia extraction, which degrades the total time of the processing procedure by over 40%. Lastra *et al.* (2015) proposed a fingerprint identification algorithm using graphic processing unit (GPU) acceleration, which can improve the speed of fingerprint identification as much as 54 times when the resource of GPU is sufficient. Xu *et al.* (2014) boosted the fingerprint matching throughput as much as 31 times by a coarse-grained parallel architecture on a 2.93 GHz Intel Xeon 5670 single core. Additionally, some works improved the speed of fingerprint identification for large databases by the message passing interface (MPI) (Peralta *et al.*, 2014). However, research on fingerprint identification systems for large-scale fingerprints in distributed systems, such as Hadoop and Spark (these platforms are more widely used than MPI in production workloads due to their stability), is still a fresh field that needs expanding.

HIPI is an image processing library, designed to be used with Apache Hadoop MapReduce. It provides a solution for how to store massive images on HDFS and make them available for future distributed computing. HIPI facilitates efficient and high-throughput image processing with MapReduce programs typically executed on a cluster. The primary input and output objects to HIPI are HIB files, which are compressed files generated by HIPI itself, capable of reducing the network transmission heavily.

MongoDB is one of the most popular and widely used databases nowadays. According to the database rank released by DB-engines in December 2015 (http://db-engines.com/en/ranking), MongoDB ranked the 4th in all databases and the 1st in NoSQL databases because of its high performance and easy-to-use. Moreover, thousands of companies including eBay and Stripe use MongoDB as their data management tool to meet the requirements like distributed storage and real-time operations (Kanoje *et al.*, 2015; Liu *et al.*, 2015).

Although distributed MongoDB itself supports a variety of technologies, such as sharding data automatically, adding nodes dynamically, processing fail-over automatically, and providing satisfying performance of balancing back-end data storage (Dede *et al.*, 2013), it lacks control of access requests of front-end mongos nodes and fails when facing some specific occasions which can lead to load imbalance of requests among front-end mongos nodes. Additionally, MongoDB's default load-balancing strategy, while guaranteeing the balance of storage amount, can easily lead to imbalance of data accessing pressure among back-end data shards, as hot data do exist in a real production environment. For example, hot data on a specific shard might suffer from great request pressure and the shard is more likely to break down. These aforementioned problems of front-end mongos nodes and back-end data shards in distributed MongoDB greatly decrease the efficiency and stability of distributed cluster in production workloads.

3 Architecture of Pegasus

The detailed explanation of Pegasus's design and implementation will be shown in the following sections and here we give a short description about the whole architecture. As shown in Fig. 1, the architecture of Pegasus can be split into two parts: the right part is a distributed feature extraction subsystem of fingerprints based on the HIPI library, and the left part is a distributed feature storage subsystem based on an optimized MongoDB cluster, which is responsible for storing fingerprints' features.

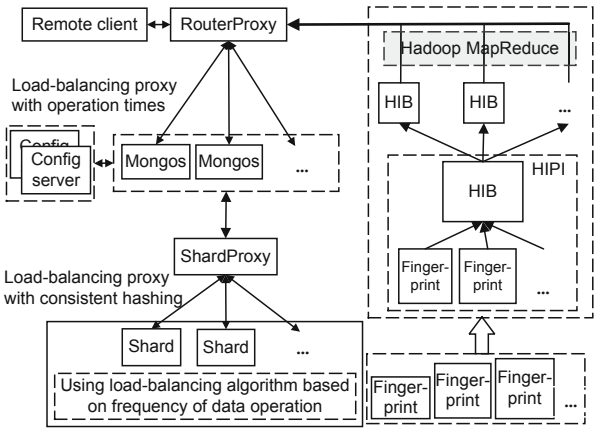

Fig. 1 System architecture of Pegasus

For the right part of Fig. 1, given a set of fingerprints, the first step is to preprocess them into the same size to simplify the distributed programming

procedure. After preprocessing, Pegasus compresses all the fingerprints into a big HIB file. The HIB file will be split into several small HIB files according to the settings of HDFS and then dispatched to the nodes of the cluster. During the map phase, on each node of the cluster, Pegasus decompresses the HIB file to obtain the inter-fingerprints and then extracts the features of each fingerprint. Later, the features of fingerprints will be collected at the reduce phase, ready for being inserted to the MongoDB.

After the aforementioned steps, for each insertion (a fingerprint's features), Pegasus sends the extracted features during the MapReduce procedure to the router server (RouterProxy: choosing a front-end mongos node in a load-balancing approach) to obtain a front-end mongos node for handling the request. Then the chosen front-end mongos node sends the features to a data shard according to an optimized load-balancing strategy. When finishing all the insertion, every fingerprint's feature will obtain its location.

When a query (fingerprint) comes from the remote client, Pegasus first extracts the fingerprint's features and then sends them to a front-end mongos node through RouterProxy. Then the front-end mongos node delivers the features to all the nodes among the cluster to obtain the target fingerprint with the highest similarity to the queried one.

4 Distributed processing of fingerprints

A fingerprint is usually featured by minutiae, including ridge endings and bifurcations. Each minutia consists of a two-dimensional coordinate and a direction as its basic information; more information can be added in terms of different situations. Extraction of minutiae generally includes a series of processes such as orientation computation, ridge extraction, minutia extraction, and filtering (Zhu *et al.*, 2005). In this study, Pegasus normalizes fingerprint images ahead of feature extraction to simplify the complexity of distributed programming.

4.1 Normalization of fingerprint images

Fingerprint images are acquired from different devices in different situations, which makes them differ in size and scale. This will increase the difficulty in distributed programming while reducing the overall performance. On the contrary, unified images will benefit the batch processing of Hadoop and HIPI. Therefore, Pegasus normalizes fingerprint images of different sizes and scales collected by different fingerprint acquisition equipment at the beginning of the whole procedure by the following steps:

Taking the fingerprints downloaded from the FVC2006 fingerprint database (Cappelli *et al.*, 2007) as an example, Pegasus first zooms them into a predetermined height (560 pixels as default). As shown in Fig. 2, the two fingerprints in Fig. 2a are zoomed to those in Fig. 2b.

Second, find the center point of each fingerprint's effective region according to the maximum pixel value of each row or column, because fingerprint images have obvious foreground and background areas (Fig. 2b). Then use the center point as the standard point to cut the fingerprint into a predetermined width (400 pixels as default). If the width of a fingerprint is larger than 400 pixels, Pegasus cuts pixels from both left and right sides of the image while keeping the center point unchanged. If the width of a fingerprint image is smaller than 400 pixels, Pegasus adds solid colored pixels, which avoids influencing the recognition of the effective region, to both sides of the image while keeping the center point unchanged.

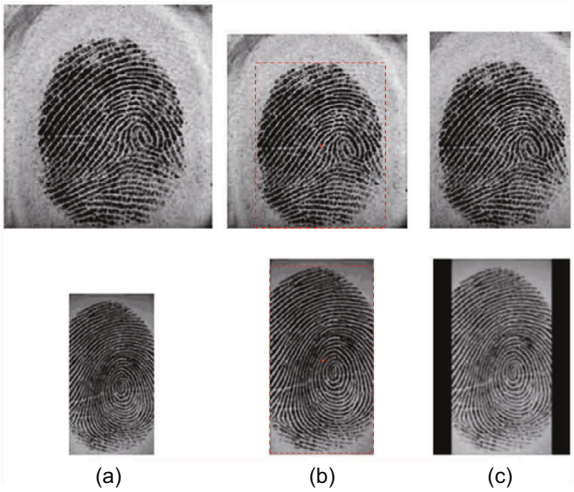

Fig. 2 Fingerprints preprocessing procedure: (a) original fingerprints; (b) center points of each fingerprint; (c) fingerprints after preprocessing. Each line shows an example. Reprinted from Zhao *et al.* (2015), Copyright 2015, with permission from Springer

4.2 Feature extraction based on HIPI

In this study, Pegasus extracts the minutia information which includes coordinate x, coordinate y, direction, and type. The type of minutiae is extracted because minutiae are always classified into two main types, ridge bifurcations and ridge endings, in most automatic fingerprint identification systems. Here Pegasus divides the types of minutiae into four categories including core point, delta point, termination point, and bifurcation point (Zhu *et al.*, 2004).

During the matching phase, two fingerprints of different types cannot belong to the same person. Therefore, Pegasus calculates the type of fingerprint into four types, including arch or tented arch, right loop, left loop, and whorl (Galar *et al.*, 2015a), which will be used for filtering out candidates during the matching phase. For load balancing during the storage phase of fingerprint's features in database, Pegasus also calculates the average inter-ridge distance (Zhu *et al.*, 2004) when extracting the orientation map because the features of fingerprints will be stored on different data shards according to their average inter-ridge distance. The final features of each fingerprint stored in MongoDB are expressed as FingerInfo in Algorithm 1.

To achieve low time expense of fingerprint processing, Pegasus implements the feature extraction procedure with HIPI. The whole procedure is separated into four steps, including preprocessing, mapping, feature extracting, and feature reducing and storing (Fig. 3).

1. HIPI transforms all fingerprints into a large HIB file, which makes the processing procedure among data nodes in cluster much faster when dealing with big files compared with small and massive fingerprints.

2. The MapReduce program on Hadoop reads the large HIB file and splits it into several small HIB files for each mapper to process (the size of each small HIB file is determined by the configuration of Hadoop).

3. Pegasus extracts the features of fingerprints, including a series of operations such as segmentation, ridge orientation enhancement, average inter-ridge distance calculation, and minutia detection.

4. When the mappers complete their tasks successfully, the reducers combine the information of fingerprints and write the features into the distributed MongoDB.

5 Load-balancing optimization of Pegasus

Load balancing is one of the most important measure standards of a distributed system, especially for high concurrence situations. The imbalance of workloads among nodes can lead to breakdown of a single node or even the whole system. Consequently, for most research, it is essential to take load balancing into consideration when constructing a

Algorithm 1 Features of fingerprint stored in MongoDB

```
1: Class FingerInfo{
2:     name: name or path
3:     avIrd: average inter-ridge distance
4:     mNum: amount of minutiae
5:     type: fingerprint's type
6:     mList: a list of minutiae
7: }
8: Class Minutia{
9:     x: value of coordinate x
10:    y: value of coordinate y
11:    orien: direction (0 to 3599)
12:    type: type of minutia
13: }
```

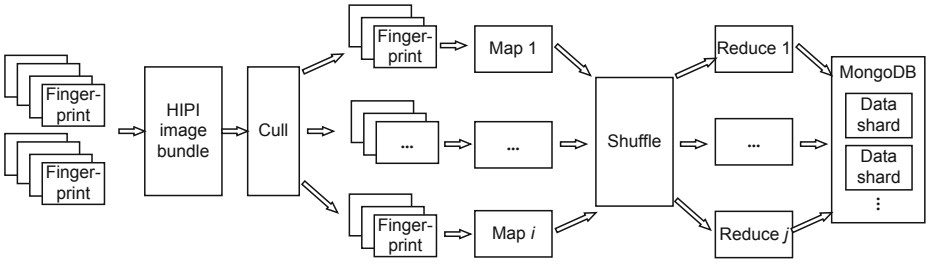

Fig. 3 Flow chart of feature extraction based on Hadoop Image Processing Interface

distributed system.

Currently, research about solving the problem of load imbalance paid more attention to the balance of data volume, which can be represented by the amount of insertion for the reason that features of different fingerprints have similar sizes (about 4 KB, including the path or name of the fingerprint, type, average inter-ridge distance, and a set of minutiae). However, researchers neglected the fact that operation load is valued most when evaluating the load-balancing performance of a distributed system of high concurrency. In this study, we propose a load-balancing strategy based on operation load for both front-end mongos nodes and back-end data shards. The 'operation load' we put forward here contains insertion, query, update, and deletion.

5.1 Load-balancing optimization of front-end mongos nodes

As the entrance from which clients can access the distributed MongoDB cluster, mongos nodes play an important role in the MongoDB cluster in terms of the cluster's stability. However, due to the fact that MongoDB does not support load balancing or automatic fail-over of front-end mongos nodes, it is a serious problem that which mongos node we should choose to query. Additionally, it is essential to react to an unpredictable breakdown properly.

To balance the access load among front-end mongos nodes, we put a front-end RouterProxy in front of the distributed MongoDB cluster for distributing queries from clients. When a query request comes, the RouterProxy will distribute it to a specific mongos node using the algorithm of consistent hashing. With this method, the RouterProxy maps all query requests and front-end mongos nodes to a circle space of the hashed value and adds virtual nodes to each of them. Consequently, all the accessing requests are distributed evenly to mongos nodes and the problem of data shard breakdown can be solved at the same time. The consistent hashing algorithm we use is shown in Algorithm 2.

Algorithm 2 Load-balancing strategy of front-end mongos nodes (consistent hashing)

```
1: Class Routers<Node>{ // data shard class
2:    nodes: map virtual nodes and real mongos nodes
3:    treeKey: map keys and real mongos nodes
4:    routers: set of real mongos nodes
5: }
6: init{ // initialization
7:    Add a mongos node to routers
8:    Use hostName of a mongos node as the input of
        hash() and map all virtual nodes to a hash ring
9: }
10: connect{
11:    key: ip address of the access requirement
12:    hash(key): calculate the hash value of each access
         requirement and map them to a hash ring
13:    getMongosToConn(key)
14:    connectMongos()
15: }
16: getMongosToConn(String key){
17:    Find the mongos node to connect by key
18: }
```

5.2 Optimization of load balancing on back-end data shards

In fingerprint identification systems, queries of fingerprints for matching are much more frequent than insertion. Although MongoDB performs well when distributing data among data shards to reach a load-balancing state (in terms of data size), the load balancing of operation is the real need of production workloads.

It is obvious that the operation load of a specific shard can be influenced by many factors besides the data volume. For instance, people from the same race or region have similar fingerprint average inter-ridge distances. If most users of an application come from the same region, then the shards that maintain the corresponding fingerprint information will suffer from much higher data accessing pressure. Therefore, the traditional load-balancing strategy will not balance the load among data shards if a fingerprint identification system involves query operations a lot.

Pegasus proposes a load-balancing strategy which can balance and migrate data dynamically according to the operation load of the MongoDB cluster to reach a load-balancing state of data operation (including insertion, deletion, update, and query). Pegasus assumes that the sizes of memory for representing different fingerprints' features are the same. Therefore, using the amount of insertion to represent the data volume in shards is credible.

5.2.1 Analysis of fingerprints' average inter-ridge distance

In the MongoDB cluster, it is essential to choose an appropriate shard key to guarantee the system with persistent scalability, for the reason that inappropriate shard keys can cause deficient partition of data collections and rapid growth of single chunk size. For example, if we choose a shard key with a small cardinal number, all the data in a collection can only be partitioned into a small number of chunks. With new records being inserted into the collection continuously, chunks of this collection will be extremely large in size quickly, and the system needs equipping new storage devices or breaks down when all the disk space runs out. If we use a shard key in ascending order, such as the default ID or timestamp field, new records will always be inserted to the last chunk. This will lead to a single non-separable data hotspot, and the shard holding this hotpot will undertake higher workload.

To avoid aforementioned undesired cases, we choose the average inter-ridge distance field as the shard key to partition the fingerprint feature collection. Average inter-ridge distance is one of the most commonly extracted features in fingerprint identification systems. It is widely distributed on the range between 57 and 126 on the FVC2006 fingerprint database that we use. To precisely design our optimized strategy, it is essential to analyze the distribution of fingerprints based on the average inter-ridge distance first.

We randomly select 5400 fingerprints from the FVC2006 fingerprint database and extract their average inter-ridge distances to analyze their distribution. We can conclude from Fig. 4 that range $[67, 80)$ has over 70% of fingerprints with the span of only a quarter. Additionally, it is clear that the closer the average inter-ridge distance to 72, the higher the percentage.

Due to the existence of hot data in the production environment, the number of fingerprints is an important reason, but not the only one, that affects the operation load. For instance, as shown by the dashed curve in Fig. 4, we assume that there is a database storing fingerprints' features of massive users. The corresponding operations on fingerprints occur most frequently when the average inter-ridge

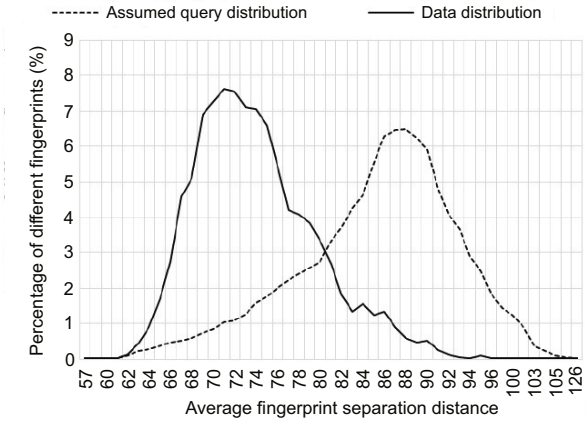

Fig. 4 Distribution of fingerprint based on the average fingerprint inter-ridge distance. Reprinted from Zhao *et al.* (2015), Copyright 2015, with permission from Springer

distance is near 90. However, as the solid curve in Fig. 4 illustrates, most fingerprints' average inter-ridge distances fall in the range of $[68, 72)$. In this situation, if we distribute data according only to the data volume, the nodes that store the fingerprints with the average inter-ridge distance around 90 will suffer from much higher pressure (data operation load) and are more likely to break down.

5.2.2 Representation of operation load

Assuming that there are n chunks in a data shard, C_i represents the ith chunk, SC_i the operation load of C_i, and SC the total operation load of this shard ($i = 1, 2, \ldots, n$). There are four types of operations on the distributed MongoDB cluster, including insertion, deletion, update, and query. We use I_i, Q_i, U_i, and D_i to represent the times of these four types of operations on C_i, respectively. Then SC_i is calculated as follows:

$$\mathrm{SC}_i = I_i + Q_i + U_i + D_i. \tag{1}$$

It is clear that four kinds of operations have different weights in different situations in terms of their differences in time consumption and frequency. Pegasus focuses more on the performance of feature extraction, which corresponds to the insertion operation, so we set the weight for insertion to a value which is larger than 1.0, while the weights of other three kinds of operations are smaller than 1.0. These weights can be changed in different situations.

Additionally, to make the whole system more flexible, Pegasus has to detect the importance of

different operations when the system is running, which means that the weight of each kind of operation should be changed dynamically. Therefore, Eq. (1) is revised as follows:

$$\mathrm{SC}_i = \mathrm{Inc}_1 \cdot I_i + \mathrm{Inc}_2 \cdot Q_i + \mathrm{Inc}_3 \cdot U_i + \mathrm{Inc}_4 \cdot D_i, \quad (2)$$

where Inc_1–Inc_4 are the corresponding weights.

In Eq. (2), the operation load of each shard SC is the sum of all the chunks' frequencies of that shard (assuming that the number of chunks on shard i is n), which can be represented as follows:

$$\mathrm{SC} = \sum_{i=1}^{n} \mathrm{SC}_i. \quad (3)$$

5.2.3 Load-balancing strategy for operation load

To finish the load balancing on operation load among shards, there are three steps to go:

Step 1: Determine the time to migrate chunks.

Step 2: Determine the chunks for migrating.

Step 3: Determine the target shard.

In this study, we implement these three steps in the following way:

For step 1, every shard maintains a variable SC and a global threshold based on the operation load that is given and available to all shards. If the gap of operation load (SC) between two different shards is larger than the global threshold, workloads on these two shards are regarded as load imbalanced and thus need to be balanced by migrating some chunks from one to the other.

For step 2, if the source shard from which chunks are migrated has more chunks, we will choose the chunks in the source shard which has the minimum SC_i to the target shard; else if the source shard has fewer chunks than the target shard, we choose the chunks in the source shard which has the maximum SC_i to the target one; if the source shard and the target shard have the same number of chunks, then choose the chunks at the top of the source shard to gain the highest migration speed. The number of chunks we choose to migrate is related to the chunks' size of the source shard and the global threshold that we set in advance. To achieve the load balancing of data operation between two shards, we need to migrate more chunks if we choose the minimum SC_i and for the maximum chunks, we need less migration time.

For step 3, migrate the chunks that are selected in step 2 to the target shard until the difference of the chunk number between the source shard and the target shard is smaller than the threshold we set in advance.

After these steps, we can reach a load-balancing state in terms of data operation load, including both access queries and insertion (data volume).

6 Coarse-grained minutiae-based fingerprint matching algorithm

Matching a fingerprint in a massive database is a significant task because the alignments are usually organized in large databases (e.g., criminal investigation system). Accordingly, many fingerprint matching algorithms have been proposed, such as minutiae-based algorithms and phase-only correlation ones (Gutiérrez *et al.*, 2014). However, minutiae-based matching algorithms are considered as the most popular and widely used ones when facing the trade-off between efficacy and efficiency (Peralta *et al.*, 2015). Xu *et al.* (2014) proposed a coarse-grained matching method which uses the coordinates and orientation of each minutia to calculate two fingerprints' similarity. Inspired by their work we propose Pegasus, and the overall matching process is shown in Algorithm 3.

The queried fingerprint and the template fingerprint from MongoDB are represented as q and temp, respectively. They are objects of FingerInfo in Algorithm 1. BOUNDD is the threshold of the Euclidean distance between two minutiae, while BOUNDA is the threshold of the orientation difference.

When a queried fingerprint q comes, Pegasus first extracts its features, including minutiae, type, and average inter-ridge distance. Then Pegasus sends the features to RouterProxy, which dispatches the query to a particular front-end mongos node for load balancing. The front-end mongos node delivers the query to the distributed MongoDB cluster to look up the fingerprint with the highest similarity. For every node in the MongoDB cluster, it filters out the fingerprints whose types are different from the queried fingerprint at the very beginning. If the fingerprints are of the same type, the matching process of two fingerprints can be detailed into the following steps:

Fig. 5a shows the minutia distribution of a

Algorithm 3 Minutiae-based coarse-grained fingerprint matching algorithm

Require: the queried fingerprint's features q and the template fingerprint's features temp in MongoDB.
Ensure: the similarity of these two input fingerprints.

```
if temp.getType()!=q.getType() then
    return 0.0;
end if
M=temp.getMNum(); // the minutia number of fingerprint temp
N=q.getMNum(); // the minutia number of fingerprint q
pair[M][2]={(−1, −1), (−1, −1), ...}; // store the value of curN and the corresponding similarity value
bottom=0; // the left/lower limit of the window on the coordinate x/coordinate y
top=0; // the right/upper limit of the window on the coordinate x/coordinate y
sum=0.0;
for m=0 to M − 1 do
    while bottom<N and q.getMList.get(bottom).getY()<temp.getMList(m).getY()−BOUNDD do
        bottom++;
    end while
    if bottom==N then
        continue;
    end if
    while top<N and q.getMList.get(top).getY()<temp.getMList(m).getY()+BOUNDD do
        top++;
    end while
    curN=−1; // the serial number of current minutia in q that acquires the minimum Euclidean distance
    curD=10 000; // the minimum Euclidean distance of the current minutia pair in q and temp
    for n=bottom to top−1 do
        dx=q.getMList.get(n).getX()−temp.getMList.get(m).getX();
        if dx<−BOUNDD || dx>BOUNDD then
            continue;
        end if
        dy=q.getMList.get(n).getY()−temp.getMList.get(m).getX();
        if dy<−BOUNDD || dy>BOUNDD then
            continue;
        end if
        dd=sqrt(dx*dx+dy*dy);
        if dd>BOUNDD then
            continue;
        end if
        ad=abs(q.getMList.get(n).getOrien()−temp.getMList.get(m).getOrien())
        if ad>1800 then
            ad=3600−ad;
        end if
        if ad>BOUNDA then
            continue;
        end if
        if curD>dd then
            curD=dd;
            curN=n;
        end if
    end for
    if curN>=0 then
        pair[m][0]=curN;
        if q.getMList.get(n).getType()!=temp.getMList.get(m).getType() then
            pair[m][1]=0.8;
        else
            pair[m][1]=1.0;
        end if
    end if
end for
for m=0 to M − 1 do
    if pair[m]>=0 then
        sum+=pair[m][1]
        for i=0 to M − 1 do
            if pair[i]==pair[m] then
                pair[i]=−1;
            end if
        end for
    end if
end for
if M<N then
    return sum/M;
else
    return sum/N;
end if
```

queried fingerprint q. The matching procedure can be transformed into the matching of q's minutiae and the stored fingerprints' features in the MongoDB cluster. When selecting a particular fingerprint's features from the MongoDB cluster (named 'temp' in Algorithm 3), Pegasus iterates the minutiae of temp to judge whether there is a minutia in q that can be matched. For example, a minutia (190, 302, 1450, 3) of temp means that the coordinate x is 190, the coordinate y is 302, the orientation is 1450, and it is a termination point. Pegasus first finds (190, 302) in fingerprint q as the triangle point (Fig. 5b) and then searches for the minutiae in q while ensuring that their Euclidean distances to the current minutia of temp are less than BOUNDD. Second, Pegasus filters out the minutiae whose orientation differences to the current minutia of temp's are larger than BOUNDA (Fig. 5c) and then searches for the target one which has the minimum orientation difference with the current minutia of temp (Fig. 5d).

Pegasus takes the type of minutiae into consideration during the matching phase. The similarity of the queried fingerprint and the template one will be much higher if they are in the same type; otherwise, the similarity is more likely to be lower (Pegasus sets the similarity between two minutiae as 0.8 when their minutia types are different).

The aforementioned procedure obtains the similarity of two fingerprints. Each node in the cluster will return the fingerprint with the local highest similarity and Pegasus chooses the candidate with the global highest similarity as the final result.

7 Evaluations

In this section, we evaluate the performance of feature extraction of fingerprint images based on HIPI and load-balancing optimization of distributed MongoDB, respectively. We also show the acceleration of Pegasus compared with DFIS during the data transmission phase between the two subsystems.

After combining the process of feature extraction with HIPI, the time consumption is only about 30% that of using the Hadoop configuration. Additionally, the load-balancing strategy of front-end mongos nodes decreases the difference of data access load to less than 5%. What is more, the optimization of MongoDB's default load-balancing strategy on back-end data shards realizes the load balance of data operation load instead of data volume, which reduces over 40% of data migration. Without additional data exchange with HDFS, Pegasus saves about 40% time consumption during the data transmission phase compared with DFIS.

7.1 Experimental evaluation of distributed processing of fingerprints

As for the performance of Pegasus in feature extraction, two experiments (fingerprints' feature extraction of Pegasus and Hadoop) have been carried out on the FVC2006 fingerprint database. We selected 14 000 fingerprints in JPG format, while in production workloads, the scale of fingerprints might be more than 10 millions or even one billion. The settings of these two experiments are as follows: (1) For

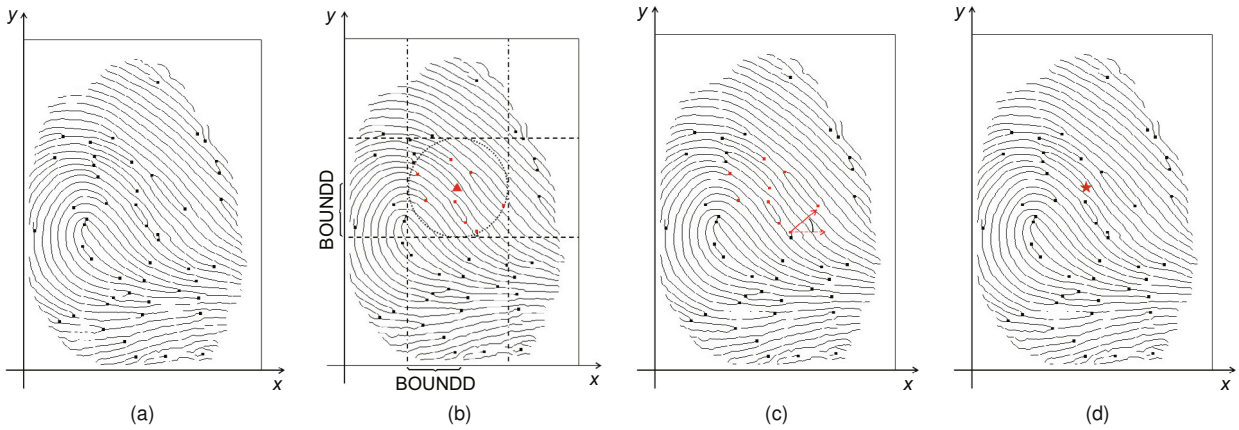

Fig. 5 Processing procedure of Algorithm 3: (a) minutia distribution of q; (b) after limiting the Euclidean distance; (c) after limiting the orientation difference; (d) selected minutia of q

feature extraction in Hadoop, one name node and several data nodes (one to eight) are contained; (2) For feature extraction in Pegasus, Hadoop and HIPI are used (this experiment has the same numbers of name nodes and data nodes as those in Hadoop).

Fig. 6 shows that with the increase of the number of data nodes, the time consumption degrades sharply. As we use the default size (64 MB) of HDFS to split data, the fingerprints will be grouped into eight blocks in the HDFS. When there is only one data node, it has to process eight chunks. For two data nodes, they have to process four chunks separately. When there are three data nodes, each of them needs to process two or three chunks. For four to seven data nodes, they have to process one or two chunks separately. For eight data nodes, each of them needs to process only one chunk. So, the experiments with the data node numbers of one, two, four, and eight are what we should consider.

We can conclude from Fig. 6 that feature exaction using MapReduce in Hadoop costs about 10 000 s and 6000 s when there are four and eight data nodes, respectively. However, Pegasus needs about 3500 s and 2000 s respectively, which reduces almost 70% time consumption.

7.2 Experimental evaluation of load balance on front-end mongos nodes

In this subsection, we deployed a MongoDB cluster of one config server and four data nodes where each one runs a mongos node at the same time. Then we wrote a MapReduce program to count the number of requests processed on each mongos node when faced with 1000 concurrent accessing requests. To evaluate whether Pegasus (using consistent hashing) truly improves the performance of MongoDB, we carried out two experiments. One used the default strategy in MongoDB when facing access requests and the other used consistent hashing of Pegasus as mentioned in Section 5.1.

Fig. 7 shows the distributions of concurrent access from multiple users among four mongos nodes. The left one is the distribution of 1000 concurrent access with the default strategy in MongoDB and the right one is using consistent hashing. As for the default strategy in distributed MongoDB, each mongos node in a server machine undertakes 34%, 27%, 24%, and 15% of 1000 concurrent requests, respectively. The difference of query load among mongos nodes is extremely high (the highest is 34% and the lowest is 15%). On the contrary, each mongos node in Pegasus undertakes 27%, 25%, 24%, and 24%, respectively.

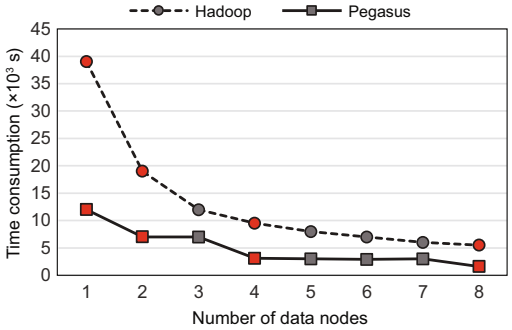

Fig. 6 Time consumption of Hadoop and Pegasus

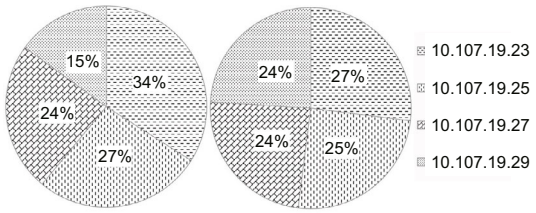

Fig. 7 Distributions of 1000 concurrent access on four mongos nodes: Hadoop (left) and Pegasus (right)

Therefore, compared with the default strategy of MongoDB, Pegasus distributes the queries among mongos nodes much more evenly and the difference of access load can be reduced to less than 5%.

7.3 Simulation of load balancing on back-end data shards

In this subsection, we show our idea about how to distribute load among back-end data shards from the view of operation. We also take the times of data migration as the standard to judge whether our optimization strategy has better performance, because migrating data among shards is the most time-consuming procedure when balancing the workload of each shard. To make the result more intuitive, we carried out two simulations and selected insertion and query (rather than all the four types of operations) as examples when computing the frequency of data operation. The weights of insertion and query were both set to 1.0 and the assumptions we made are as follows:

(A1) There were four shards in the cluster.

(A2) The size of an item which needs inserting was 1 MB.

(A3) If a chunk was larger than 200 MB, then it would be split into several smaller chunks.

(A4) The thresholds of load imbalance were set to 400 MB for shard size and 400 times for operation (query and insertion).

(A5) Since the chunk size was 200 MB and the thresholds were both 400 (MB or times), each time we would migrate chunks with the size of 200 MB.

(A6) 10 000 insertions and 10 000 queries occurred with the same probability of 50%.

(A7) The weights of insertion and query were both 1.0.

According to the distribution of fingerprints based on the average inter-ridge distance in Fig. 4, we formulate its distribution with Gaussian distribution as follows (Zhao *et al.*, 2015):

$$f(x) = 323\exp\left[-\left(\frac{x-70.9}{5.087}\right)^2\right] + 175.3\exp\left[-\left(\frac{x-77.1}{8.026}\right)^2\right]. \quad (4)$$

To evaluate our optimization strategy, we construct a formula based on the query frequency and average inter-ridge distance, satisfying Possion distribution, which is similar to the dotted curve in Fig. 4. Its fitting formula with Gaussian distribution is as follows (Zhao *et al.*, 2015):

$$f(x) = 289\exp\left[-\left(\frac{x-105.1}{13.01}\right)^2\right] + 162\exp\left[-\left(\frac{x-89.17}{29.21}\right)^2\right]. \quad (5)$$

Based on assumptions (A1)–(A7), we carried out two simulations, each of which was repeated 20 times to obtain the average result. The experiments were constructed in different situations:

1. The default migration strategy in distributed MongoDB migrated the chunks at the top of the source shard if the difference of the chunk number between the source shard and target shard was larger than the threshold set in advance.

2. The optimized migration strategy based on data operation load migrates chunks according to the insertion (or data size) and query as explained in Section 5.2.

After these simulations, we counted the times of insertion and query on each shard and obtained the histograms which show the distribution of query, insertion, and operation load of these two simulations.

Fig. 8 shows the distributions of query time, data size, and the operation load of each shard with the default strategy in MongoDB. During the simulation, the default strategy takes 6.3 times chunks' migration on average to get the data balanced, and we can conclude from Fig. 8 that the MongoDB's default strategy balances the load of each shard quite well in terms of data size. However, the query load varies a lot among shards, as MongoDB neglects the query operation.

Fig. 9 shows the distributions of query times, data size, and operation load of each shard with the optimized strategy. During the simulation, the optimized strategy takes 3.2 times chunks' migration on average to get the data balanced and we can conclude

Fig. 8 Distributions of data size and query time with the default strategy in MongoDB. Reprinted from Zhao *et al.* (2015), Copyright 2015, with permission from Springer

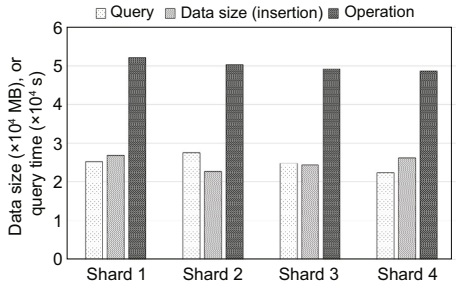

Fig. 9 Distributions of data size and query time with the optimized strategy in MongoDB. Reprinted from Zhao *et al.* (2015), Copyright 2015, with permission from Springer

from Fig. 9 that the optimized strategy reduces difference in both data size and queries among shards. Although the data distribution is not as good as that of the default strategy in MongoDB in terms of data size, the queries and total operation load are much more balanced.

We can conclude from Figs. 8 and 9 that the default load-balancing strategy in MongoDB can achieve the load balance of data size pretty well. In contrast, the optimization strategy of Pegasus balances both the data size and data operation load, even though the performance of load balance in terms of data size is not as good as that of MongoDB's default strategy. What is more, Pegasus needs less migration of chunks.

Furthermore, we choose another steeper formula of query distribution as Eq. (6) and keep the insertion distribution unchanged at the same time (Zhao *et al.*, 2015):

$$f(x) = 520\exp\left[-\left(\frac{x-105}{13.82}\right)^2\right] + 189\exp\left[-\left(\frac{x-82.92}{21.92}\right)^2\right]. \quad (6)$$

After the same simulation, we obtain that the average migration times of the new distribution (much steeper one) is 5.35 times using MongoDB's default strategy and 3.0 times using the optimized strategy of Pegasus. Because insertion and query occur randomly, the results of our simulations are extensible. We conclude that when the distribution of data operation load is different from that of data volume based on the average inter-ridge distance, Pegasus can achieve a better performance no matter the query's distribution curve is flat or steep.

7.4 Analysis of Pegasus's load balancing among back-end data shards

7.4.1 Pegasus needs less migration of chunks

In MongoDB's default strategy of data migration, each data shard maintains a stack to store its data chunks. If the difference of data chunks' number between two shards exceeds a predefined threshold, the chunks at the top of the stack in the source data shard will be migrated. However, this strategy will cause the following obvious problem:

If the chunks moved from the source data shard contain the key range of low occurrence frequency (Fig. 4) and the source data shard keeps the chunks with the key range of high occurrence frequency, then the data load of chunks to be moved will suffer from fewer insertions in the future (the default migration strategy of MongoDB is based on the data volume, so insertion is the only operation that can affect data distribution). Therefore, although the chunks have been migrated from the source data shard, the source data shard will become overloaded again soon.

To conquer this problem, Pegasus first compares the number of chunks (represent the data volume or insertions) in the source and target data shards before migration to ensure that data shards can maintain a longer stable state after migration. There are two strategies in the migration procedure:

If the source data shard has more chunks (more data insertion), which means that other operations (deletion, update, and query) of the source data shard, especially query (fingerprint identification systems have more queries than insertions), play a less important role when judging whether there should be a migration between the source data shard and the target one (because insertion plays an essential role during the judgement), then it is wise to keep those chunks with high operation frequency, so that this data shard will have more operations in the future, or this shard will have less operation load in the future and become the target one during the upcoming migration processes soon. Based on this strategy, Pegasus moves the chunks with the smallest SC_i, which makes more data chunks move to the target data shard and keeps those data chunks with more operation load.

If the source data shard has a smaller number of chunks, indicating that the data operation frequency on the source data shard is higher than that on the target one, then Pegasus moves the data chunks with the largest SC_i. In doing so, Pegasus can quickly migrate the data chunks with high operation frequencies to the target one and keeps more data volume on the source data shard. By moving data chunks of high operation frequencies, the source data shard will suffer from less operation load in the future and will be more likely to keep a long-time load-balancing state.

7.4.2 Pegasus keeps load balancing of data volume

For Pegasus's data chunk migration, if the source data shard has more chunks, it chooses the chunks with the smallest SC_i to move, which means more chunks will be moved to the target data shard and this will benefit the load balancing of data volume among data shards a lot. On the other hand, if the source data shard has fewer data chunks, it chooses the chunks with the largest SC_i to move, which means fewer chunks will be moved.

Therefore, Pegasus can ensure that each data shard of the MongoDB cluster has data chunks of both low operation frequency and high operation frequency. Data chunks are moved according to their SC_i and SC of the source data shard, which keeps a long-time load balancing of data volume as well.

7.5 Pegasus's acceleration compared with DFIS

In our previous work DFIS, the feature extraction subsystem and database storage subsystem were designed as two separated subsystems, and additional HDFS I/O (input/output) for data transmission was required. Fingerprints' features need to be output to HDFS first, and then be read and written to MongoDB. Pegasus accelerates this procedure by combining two subsystems more compactly and eliminating additional HDFS access. The acceleration is achieved thanks to the usage of the MongoDB-Hadoop connector, which can use either HDFS or MongoDB as the source or destination of MapReduce.

To depict the improvement visually, we carried out two groups of experiments on three datasets. For these experiments, we selected 10 000, 100 000, and 1 000 000 fingerprints from the FVC2006 fingerprint database, respectively (we used multiple copies because the fingerprints are not sufficient). Then we calculated the runtime of DFIS and Pegasus when transferring the features between the distributed feature extraction subsystem and the distributed feature storage subsystem.

Fig. 10 illustrates the runtime comparison of data transmission between the aforementioned two subsystems on different datasets. It can be seen that Pegasus realizes about 40% time reduction when transmitting data between these two subsystems, which contributes to the whole performance enhancement tremendously.

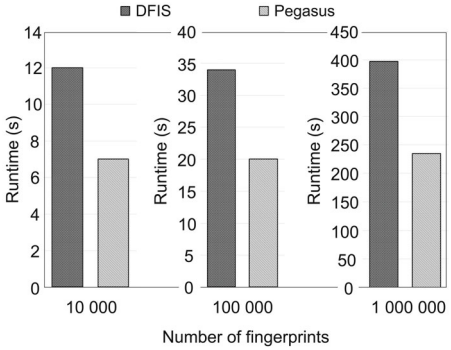

Fig. 10 Runtime comparison of data transmission between two subsystems on different datasets

8 Conclusions and future work

Aiming at handling large-scale feature extraction, feature storage, and concurrent access of fingerprints, we have designed a distributed and load-balancing fingerprint identification system, Pegasus. Pegasus uses HIPI to conquer high time consumption when processing massive fingerprints with cluster computing systems. Compared with the Hadoop platform, it degrades the time consumption by almost 70%.

As for the load balancing of the MongoDB cluster, Pegasus adopts consistent hashing to distribute the concurrent requests. It degrades the difference of access requests to less than 5%, which improves the stability of the distributed MongoDB tremendously. What is more, Pegasus uses a load-balancing strategy from the view of operation level, which includes insertion (data volume), deletion, update, and query. The corresponding simulations and experiments have proved that it can use less data migration to gain a more reasonable distribution of operation load among data shards, especially when the distribution of operation load is different from that of data volume.

Additionally, Pegasus saves the features of fingerprints extracted into the distributed feature storage subsystem immediately. Without additional data operations (load and write) with HDFS, Pegasus reduces as much as 40% time consumption compared with DFIS.

In the future, it is worth implementing the

feature extraction procedure on Spark (including the implementation of HIPI on Spark), which may boost the feature extraction procedure further. Also, we will try to implement the load-balancing strategy of back-end data shards in a real MongoDB cluster to make it available for more users.

References

Cappelli, R., Ferrara, M., Franco, A., *et al.*, 2007. Fingerprint verification competition 2006. *Biomet. Technol. Today*, **15**(7-8):7-9. http://dx.doi.org/10.1016/s0969-4765(07)70140-6

Dede, E., Govindaraju, M., Gunter, D., *et al.*, 2013. Performance evaluation of a MongoDB and Hadoop platform for scientific data analysis. Proc. 4th ACM Workshop on Scientific Cloud Computing, p.13-20. http://dx.doi.org/10.1145/2465848.2465849

Galar, M., Derrac, J., Peralta, D., *et al.*, 2015a. A survey of fingerprint classification part I: taxonomies on feature extraction methods and learning models. *Knowl.-Based Syst.*, **81**:76-97. http://dx.doi.org/10.1016/j.knosys.2015.02.008

Galar, M., Derrac, J., Peralta, D., *et al.*, 2015b. A survey of fingerprint classification part II: experimental analysis and ensemble proposal. *Knowl.-Based Syst.*, **81**:98-116. http://dx.doi.org/10.1016/j.knosys.2015.02.015

Gutiérrez, P.D., Lastra, M., Herrera, F., *et al.*, 2014. A high performance fingerprint matching system for large databases based on GPU. *IEEE Trans. Inform. Forens. Secur.*, **9**(1):62-71. http://dx.doi.org/10.1109/tifs.2013.2291220

Hong, L., Wan, Y., Jain, A., 1998. Fingerprint image enhancement: algorithm and performance evaluation. *IEEE Trans. Patt. Anal. Mach. Intell.*, **20**(8):777-789. http://dx.doi.org/10.1109/34.709565

Indrawan, G., Sitohang, B., Akbar, S., 2011. Parallel processing for fingerprint feature extraction. Proc. Int. Conf. on Electrical Engineering and Informatics, p.1-6. http://dx.doi.org/10.1109/iceei.2011.6021606

Kanoje, S., Powar, V., Mukhopadhyay, D., 2015. Using MongoDB for social networking website deciphering the pros and cons. Proc. Int. Conf. on Innovations in Information, Embedded and Communication Systems, p.1-3. http://dx.doi.org/10.1109/iciiecs.2015.7192924

Lastra, M., Carabaño, J., Gutiérrez, P., *et al.*, 2015. Fast fingerprint identification using GPUs. *Inform. Sci.*, **301**:195-214. http://dx.doi.org/10.1016/j.ins.2014.12.052

Li, J., Li, D., Ye, Y., *et al.*, 2015. Efficient multi-tenant virtual machine allocation in cloud data centers. *Tsinghua Sci. Technol.*, **20**(1):81-89. http://dx.doi.org/10.1109/tst.2015.7040517

Liu, C., Ouyang, K., Chu, X., *et al.*, 2015. R-memcached: a reliable in-memory cache for big key-value stores. *Tsinghua Sci. Technol.*, **20**(6):560-573. http://dx.doi.org/10.1109/tst.2015.7349928

Mader, K., Donahue, L., Müller, R., *et al.*, 2014. High-throughput, scalable, quantitative, cellular phenotyping using X-ray tomographic microscopy. Proc. 2nd Int. Work-Conf. on Bioinformatics and Biomedical Engineering, p.1483-1498.

Malakar, R., Vydyanathan, N., 2013. A CUDA-enabled Hadoop cluster for fast distributed image processing. Proc. National Conf. on Parallel Computing Technologies, p.1-5. http://dx.doi.org/10.1109/parcomptech.2013.6621392

Peralta, D., Triguero, I., Sanchez-Reillo, R., *et al.*, 2014. Fast fingerprint identification for large databases. *Patt. Recog.*, **47**(2):588-602. http://dx.doi.org/10.1016/j.patcog.2013.08.002

Peralta, D., Galar, M., Triguero, I., *et al.*, 2015. A survey on fingerprint minutiae-based local matching for verification and identification: taxonomy and experimental evaluation. *Inform. Sci.*, **315**:67-87. http://dx.doi.org/10.1016/j.ins.2015.04.013

Plugge, E., Hawkins, D., Membrey, P., 2010. The Definitive Guide to MongoDB: the NoSQL Database for Cloud and Desktop Computing. Apress.

Shu, Y., Gu, Y.J., Chen, J., 2014. Dynamic authentication with sensory information for the access control systems. *IEEE Trans. Parall. Distr. Syst.*, **25**(2):427-436. http://dx.doi.org/10.1109/TPDS.2013.153

Sweeney, C., Liu, L., Arietta, S., *et al.*, 2011. HIPI: a Hadoop Image Processing Interface for Image-Based MapReduce Tasks. MS Thesis, University of Virginia, USA.

Xu, J., Jiang, J., Dou, Y., *et al.*, 2014. A low-cost fully pipelined architecture for fingerprint matching. Proc. 12th Int. Conf. on Signal Processing, p.413-418. http://dx.doi.org/10.1109/icosp.2014.7015039

Zhang, Z., Li, D., Wu, K., 2016. Large-scale virtual machines provisioning in clouds: challenges and approaches. *Front. Comput. Sci.*, **10**(1):2-18. http://dx.doi.org/10.1007/s11704-015-4420-7

Zhao, Y., Zhang, W., Li, D., *et al.*, 2015. DFIS: a scalable distributed fingerprint identification system. Proc. 15th Int. Conf. on Algorithms and Architectures for Parallel Processing, p.162-175. http://dx.doi.org/10.1007/978-3-319-27137-8_13

Zhu, E., Yin, J., Zhang, G., 2004. Computation of fingerprint inter-ridge distance. *J. Microelectron. Comput.*, **21**(10):7-9 (in Chinese).

Zhu, E., Yin, J., Zhang, G., 2005. Fingerprint matching based on global alignment of multiple reference minutiae. *Patt. Recog.*, **38**(10):1685-1694. http://dx.doi.org/10.1016/j.patcog.2005.02.016