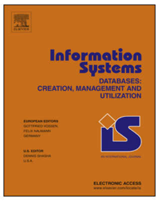
Contents lists available at ScienceDirect

Information Systems

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



Q2 Information system for image classification based on frequency curve proximity

L. Sánchez[a], Javier Alfonso-Cendón[a,*], Tiago Oliveira[b],
Q1 Joaquín B. Ordieres-Meré[c], Manuel Castejón Limas[a], Paulo Novais[b]

[a] University of León, Leon, Spain
[b] University of Minho, Braga, Portugal
[c] Polytechnic University of Madrid, Madrid, Spain

A R T I C L E I N F O

Article history:
Received 8 June 2015
Received in revised form
29 August 2016
Accepted 31 August 2016

Keywords:
Information system
Similarity search
Frequent itemset mining
Metadata
Image classification

A B S T R A C T

With the size digital collections are currently reaching, retrieving the best match of a document from large collections by comparing hundreds of tags is a task that involves considerable algorithm complexity, even more so if the number of tags in the collection is not fixed. For these cases, similarity search appears to be the best retrieval method, but there is a lack of techniques suited for these conditions. This work presents a combination of machine learning algorithms put together to find the most similar object of a given one in a set of pre-processed objects based only on their metadata tags. The algorithm represents objects as character frequency curves and is capable of finding relationships between objects without an apparent association. It can also be parallelized using MapReduce strategies to perform the search. This method can be applied to a wide variety of documents with metadata tags. The case-study used in this work to demonstrate the similarity search technique is that of a collection of image objects in JavaScript Object Notation (JSON) containing metadata tags.

© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

Due to the current diversity and availability of image capturing devices, such as digital cameras, digital scanners and smartphones, and the use of the Internet to disseminate content, the size of digital image collections is continuously increasing. The predictions from a technical report from the International Data Corporation [1] point towards a growth of digital content from 130 exabytes to 40,000 exabytes, between 2005 and 2020. This implies a heavy investment in Information Technology hardware, software, services, telecommunications, and staff, in short, of all the components that make up the infrastructure of the digital universe. Most of this information is produced by average consumers in their interaction with social media, by sending camera phone images and videos between devices and around the Internet, and so on [2]. While the information holding potential analytical value is growing at an unbelievable rate, only a small fraction of this information has been explored. The effective management of these collections has become a necessity for both companies and the general public.

Classical database management systems (DBMSs) are designed to handle data objects that have a pre-established structure. Normally, this structure is acquired by treating every feature of a data object as an independent dimension, and then building representations in the

* Correspondence to: Dpto. Ingenierías Mecánica, Informática y Aeroespacial, Escuela de Ingenierías Industrial e Informática, Universidad de León, 24071 León, Spain.
E-mail addresses: lidia.sanchez@unileon.es (L. Sánchez), javier.alfonso@unileon.es (J. Alfonso-Cendón), toliveira@di.uminho.pt (T. Oliveira), j.ordieres@upm.es (J.B. Ordieres-Meré), manuel.castejon@unileon.es (M.C. Limas), pjon@di.uminho.pt (P. Novais).

http://dx.doi.org/10.1016/j.is.2016.08.001
0306-4379/© 2016 Elsevier Ltd. All rights reserved.

form of records. These records are then stored according to a certain database model which can be relational, object-oriented, object-relational, hierarchical, etc. However, these models require that the data objects have a fixed, and typically reduced, number of features because queries are usually performed by exact matching, partial matching, and joining applied to some of the features. Yet, there are applications that demand the use of data with a simplified structure, and, thus, less organized and precise [3]. The problem with this type of data is that it is nearly impossible to order it and it is not meaningful to perform equality comparisons on it. For these cases, proximity, or similarity, is a more suitable search criterion. Similarity search is a central component to content-based retrieval in multimedia database systems. It is a general term that includes a wide range of techniques whose main goal is normally one of the following [4,5]: (1) to find objects whose feature values fall within a range of distance, using a defined metric, from a query object (range queries); (2) to find a certain number of objects whose features are the closest to an object query (nearest neighbor queries); and (3) to find pairs of objects within the same set which are similar to each other.

As such, efficient search and retrieval mechanisms are a basic need in systems that deal with these collections in a wide variety of domain applications. Photography, fashion, crime prevention, architecture, publishing, journalism and academic research itself are only a few examples of domains where image search systems are necessary. However, going through large collections of documents is a hazardous task and involves the use of expensive computational resources. There is a clear need for an object search method that is quick, lightweight, and easy to apply to large item collections.

Metadata is normally referred to as *data about data*. It provides additional information that supplements the content of images. As such, it has become a powerful mechanism to search through the content of image libraries and other digital media such as audio and video [6]. Using metadata is considered advantageous because it is still impractical, namely in the field of digital photography, to organize and query images based on millions of image pixels. Considering this, it is preferable to use metadata properties describing what the picture represents and details (where, when and how) of its capture.

The premise of this work is that the structural and descriptive metadata of an image can provide useful cues, independent of the captured scene content, for image retrieval and matching. To test this hypothesis, one developed an algorithm that constructs characteristic curves of image objects by analyzing all the metadata tags in a document. Using these curves, the algorithm can perform fast searches in the document database and retrieve a list of images sorted by proximity to a given one. The advantage of the algorithm lies in being possible to group similar objects in order to determine if different objects have the same origin. This kind of relationship may be a great advantage in order to know more about the history of an image, to know if it has been modified or tampered with. The setting used to test the algorithm includes a collection of JavaScript Object Notation (JSON) objects containing the metadata tags of images in multiple formats. JSON is an emerging data transfer format and it is used as an access method in many NoSQL database [7], which are an example of the systems that house the simplified and less structured data mentioned earlier. NoSQL provides horizontal scaling and, thus, in particular conditions enables a faster retrieval. The computation can be divided in concurrent tasks across distributed machines. To achieve this, these systems have to relax some of the characteristics of traditional DBMSs, one of which is data structure. At the same time, this is also a desirable feature for certain data types, such as images, which come in multiple formats, each one of them with different tags. The algorithm was implemented using Go [8], a programming language developed by Google that provides more facilities for the implementation of concurrency and parallelism in order to get the most out of multicore and networked machines.

The paper is organized as follows. Section two provides related work in the fields of similarity search, itemset mining, and image metadata. Section three is considered a materials and methods section which has a description of the technique and search strategy, of how the frequency curves for the documents are constructed, and of how to perform a search query using the developed algorithm. In this section, there are also results that demonstrate their effectiveness. Section three features a discussion where the main strengths and limitations of the approach are highlighted. Finally, in section five conclusions are drawn about the main contributions of the work.

2. Related work

This section provides information on the three main topics of this work: similarity search, frequent itemset mining (FIM), and image metadata. Given the vastness of the work developed in similarity search, only the aspects and approaches that bear a resemblance or can offer a good counterpoint to the approach followed herein will be mentioned. Central to this work is also discovering which features from a given set in an object collection are the most important for conducting similarity search queries, thus the inclusion of FIM in the topics of interest. The section ends with a description of what metadata is, its purposes and its issues.

2.1. Similarity search

Similarity search has established itself as one of the fundamental paradigms in modern applications. This is an important task when trying to find patterns in applications, involving the exploration of data such as images, videos, time series, text documents, and so forth.

In essence, it consists in a problem of finding, within a set of objects, those which are more similar to a given query object. Normally, data collections are treated as metric objects, which brings significant advantages because many data classes and information-seeking strategies conform to the metric view. There are four fundamental aspects of similarity search: the distance measure, the type of query, the partition principle, and the index structure [9].

The similarity is usually assessed by a distance function, meaning that low values of distance correspond to high degrees of similarity. The obvious advantage of this is that the results can be ordered according to their estimated relevance. A nice abstraction for nearness or proximity is the metric space, also called distance space [10,11]. It corresponds to the mapping of an object to a set of values in a distance measure, within a certain domain. A special case of the metric space is the coordinate space, in which the objects are represented as vectors. There are works that exploit different types of distance measures, namely the Minkowski distances, the quadratic form distance, the edit distance, the Jaccard's coefficient, and the Hausdorff distance [9]. Minkowski distances [12] describe a family of distance functions, referred to as L_p metrics, that depend on a parameter p. They are defined for n-dimensional vectors of real numbers as follows in Eq. (1).

$$L_p\left[(x_1,\dots,x_n),(y_1,\dots,y_n)\right]=\sqrt[p]{\sum_{i=1}^{n}|x_i-y_i|^p} \tag{1}$$

Minkowski distances are appropriate when the objects in a collection are represented as vectors. Two of the most studied distance functions are the Manhattan distance (L_1) and the Euclidean distance (L_2) [9].

As mentioned earlier, there are three types of similarity search queries. However, this work focuses on one in particular, the nearest neighbor queries ($kNN(q)$). A $kNN(q)$ query retrieves the k nearest neighbors of the object q. To achieve this and search the metric space, one can follow different strategies, which are mirrored in the partition principle used in the search space. Partitioning divides the search space into sub-groups so that only some of these groups are searched when a query is given. Partition principles can be roughly divided into: ball partitioning, generalized hyperplane partitioning [13] and excluded middle partitioning [14]. In ball partitioning, an object is selected from the collection to be used as a pivot, then, using a certain distance value to the other objects, a spherical cut is made, dividing the search space into two subsets. In hyperplane partitioning (also called clustering-based partitioning), the space is partitioned into two sets of disjoint clusters where each one is represented by a cluster center. As for excluded middle partitioning, it is an extension of ball partitioning, but instead of splitting the space into two subsets, it splits the space into three subsets. In this case, when an object query is near the partitioning threshold, it implies accessing both ball-partitioned subsets.

The search strategy depends highly on how the data is partitioned. The most basic of strategies, yet not always inefficient, is to scan the data sequentially. However, depending on the available resources and the size of the search space, it may result in the computation of a significant number of distance measures and ranking algorithms that are computationally expensive. What most approaches do is to focus on the construction of search/index structures for performing similarity search over complex structures.

There are tree-like index structures that use ball partitioning to get subregions of space, each containing objects stored in a small number of disk blocks. Given an object as a query, the object is first placed in a sub-region of the tree, then a nearest-neighbor query is performed to get the regions where the closest objects may reside. Examples of tree-like index structures include the Burkhard-Keller Tree [15], the Vantage Point Tree [16], the Multi-way Vantage Point Tree [17], the Excluded Middle Vantage Point Forest [14], and other variations. The common disadvantage to these structures lies in the performance bottleneck when trying to locate the sub-region of the object query, because the tree structure may be too big to fit in main memory, which implies having part of it on disk and performing numerous input/output operations. Also, the number of neighboring sub-regions can grow exponentially with respect to the dimensionality, resulting in another performance bottleneck [18,19].

The Bisector Tree [20] was the first index structure to implement generalized hyperplane partitioning. It recursively partitions each cluster into two clusters, based on two pivots chosen initially at random. The covering radii, which are the maximum distances between the pivots and any object in their subtrees, are used to prune the branches. The Voronoi Tree [21] is an improved version of this structure which uses two or three pivots in each internal node. The Generalized Hyperplane Tree [13] follows the same principle, but with a different criteria for pruning. Instead it uses the hyperplane between pivots to decide which subtrees to visit. The difficulty with these methods lies in achieving balanced clusters that can efficiently cut down search costs.

To answer the dynamic nature of real databases, the M-Tree [22] was developed as a structure for efficient secondary memory storage, supporting dynamic insertions and deletions of objects. Unlike other structures, the M-Tree is built bottom-up and maintains the same size in all sub-trees, because the tree is balanced. The M-Tree is very popular and there are numerous extensions of this pattern such as the Multi-Way Insertion Algorithm [23] and the Slim Tree [24], just to name a few.

The performance of $kNN(q)$ search algorithms is usually dictated by two criteria: the number of distance computations and the input/output costs for processing nearest neighbor queries on distance data [9].

2.2. Frequent itemset mining

FIM techniques can be used in a wide variety of applications, including association rule mining, indexing, classification, and clustering. They are a form of unsupervised learning used to extract information from databases based on events that occur frequently, as a way to capture meaning beyond that of individual features. FIM techniques are more suitable for poorly understood problem domains. Formally, the problem is defined as follows [25]. Let I be a set of items $o_1, o_2,\dots,o_d$. A subset of I is called an itemset. A *transaction dataset* is a collection of itemsets, $D=\{t_1,\dots,t_n\}$, in which $t_i \subseteq I$. For any itemset α, the transactions that contain α are written as $D_\alpha=\{t_i|\alpha\subseteq t_i \text{ and } t_i\in D\}$. In a transaction dataset D, an itemset α is frequent if

$|D_\alpha|/|D| \geq \sigma$, where $|D_\alpha|/|D|$ is the support of α in D, represented as $\sigma(\alpha)$, and σ is the minimum support threshold, $0 \leq \sigma \leq 1$. All frequent itemsets share the Apriori property, which states that any subset of a frequent itemset is frequent. Since the number of features in big datasets is high, it is beneficial and computationally less expensive to mine frequent itemsets than to check for the effect of combinations between all the features.

Arguably, the most influencing works in this field are those of [26,27], and [28]. The first one describes the well-known Apriori algorithm, which focuses on the problem of discovering association rules between items in a large database of sale transactions. Apriori [26] counts items by making passes over the transaction dataset D, thus finding the most frequent items. Afterwards, it generates candidate itemsets by combining the frequent items. The first group of candidate itemsets have length 2 and their support is calculated with another pass over D. The process repeats itself for itemsets with increasing cardinality until k-length itemsets are found. The main advantage of the algorithm is providing a performance gain by significantly reducing the search space. There are a lot of other algorithms proposed after the introduction of Apriori, they resulted from the optimization of certain steps within the structure of the algorithm. Performance is majorly dictated by the support counting procedure, so research has focused mainly on that aspect, which resulted in algorithms such as AprioriTid, AprioriHybrid, Direct Hashing and Pruning (DHP), and so forth. All these algorithms are assessed in [29], a fairly comprehensive survey. To sooth the high input/output overhead of scanning large databases with Apriori, new FIM implementations were developed, many of which are based on parallel algorithms derived from Apriori [30]. An alternative is the Eclat algorithm [27]. It performs parallel mining of association rules by traversing a prefix tree in a *depth-first* manner in order to find frequent patterns. If a path in a prefix for an itemset in the tree is infrequent, it concludes that all of its subtrees are also infrequent and they are immediately pruned. On the other hand, if an itemset is frequent, it is treated as a prefix and extended to form new itemsets. Eclat uses a vertical database format for the faster computation of supports. Another depth-first algorithm is FP-growth [28]. It uses a combination of the vertical and horizontal database layout to store the database in main memory. This layout is a frequent pattern tree (FP-Tree) structure, which is an extended prefix tree for storing condensed information about frequent patterns. The gain in efficiency for this algorithm is achieved by compressing a database (the transactions supporting an itemset) into the mentioned FP-tree, avoiding costly database scans. The overall objective is to store the most frequent patterns closer to the root, since they are the ones which are most likely to be shared, thus obtaining a compact structure which is computationally cheaper to traverse. There is a vast number of variations of these algorithms and equally vast performance studies about which is the best FIM algorithm, but the main conclusion to draw from them is that the choice of which algorithm to use is mainly determined by implementation, data set, and parameter settings [31].

FIM has been mostly applied in tasks of association rule mining and classification. There are works in which FIM is applied to the extraction of high level features to capture more discriminative information. In [32] the objective is finding the most suitable patterns for image classification from a set of features in a *bag-of-visual-words* model, which also corresponds to a histogram representation. After finding the most relevant patterns, the images are represented in the form of *frequent local histograms*. Then, making use of a standard histogram intersection kernel, supervised learning is performed in order to derive classes for the posterior classification of new images. There are two central concepts to this work which are also applicable to similarity search: *discriminative power* and *pattern frequency*. These issues are discussed in more detail in [33], which discusses the choice of the best value for the minimum support threshold. While being frequent, the itemsets also have to contain enough different values to allow for distinctions between objects. The authors achieved this by building a connection between pattern frequency and discriminative measures such as *information gain* and *Fisher's score*.

2.3. Image metadata

As pointed out earlier, with the low cost of technology capable of producing and disseminating digital images, the amount of digital content being produced is rapidly increasing. Storage capacity has accompanied this development and, now, there is practically no limit to the amount of digital images one is able to keep. When there is no information about the content of images, the only way to search through them is through metadata [6].

According to the National Information Standards Association (NISO), *"metadata is structured information that describes, explains, locates, or otherwise makes it easier to retrieve or manage an information resource"* [34]. Cameras capture device metadata while taking pictures. Then, operating systems and other software tools use this metadata to build catalogs and provide effective searching functionalities. Moreover, users can add their own metadata to images to improve these functionalities. However, the existence of different metadata standards leads to interoperability issues when dealing with different software tools. Even though most metadata properties are unique, there are a number of them that are specific of certain standards and may overlap across other standards [6].

Digital images come in a variety of file formats. Formats such as JPEG, PNG, TIFF and PSD, just to name a few, have distinct rules on how metadata is stored within a file. They may follow different format standards for metadata containers such as the Exchangeable Image File Format (EXIF), the Adobe (Extensible Metadata Platform) XMP, the International Press Telecommunications Council (IPTC), among others [35]. These standards define ta for recording a wide variety of information such as when and how the images were created, digital camera specifications, and other technical information (e.g. lighting conditions). Different applications and devices have chosen to follow different policies regarding metadata storage, which causes a great variability in the metadata tags an image may have.

There are a few works of more elaborate search using image metadata. Yee et al. [36] propose an interface that makes use of hierarchical faceted metadata that allows a user to visually search for images, and, while searching, the interface displays dynamically generated query previews. Korenblum et al. [37] developed a system for managing biomedical image metadata. The type of metadata used in this work is semantic, meaning that it is mostly composed of textual annotations about image content. The system receives inputs in the form of standard-based metadata files through a Web service, then it parses and stores the metadata in a relational database. When querying the system, the search engine searches for metadata tags similar to the ones introduced, and, when it finds close matches, it automatically renders 2D regions of interest stored as metadata. The system is available through a Web Application. Zhang et al. [38] developed a vertical image search engine based on location metadata, optimized for the retrieval and ranking of images from locations specified by user queries. All these works are good examples of how metadata can be used to improve querying capabilities. However, concerning this literature review, it seems that there are no significant works using this kind of metadata for similarity search, as most of the existing work focuses mainly on content-based image retrieval.

Depending on the objectives of the query, the algorithm presented herein may be useful if one wants to trace back the origin of an image or check if the image document being submitted as a query is the modified version of an existing image. Given the wide variety of metadata tags, it is very difficult to discover those which are the most important in computing similarities. There is also the issue with the different number of metadata tags each image may have, which makes the similarity search even more difficult.

3. Material and methods

The algorithm for classification and retrieval consists of a sequence of procedures. The first of which is to build a dictionary of metadata tags in a document collection and to assign uniqueness grades to those tags. After this, the algorithm constructs tag groups that include the most frequent tags in the image collection used as sample. Once the groups are retrieved, the next step is to build the characteristic curves for the objects in the collection. These curves will be used to calculate the distance of an object in the collection to an object in a search query. The following subsections represent and explain the different procedures performed by the algorithm.

3.1. Uniqueness grade for metadata tags

Not all the metadata tags have the same relevance when comparing image objects. For instance, in an image, the metadata tags *FlashFiring* and *FlashFunction* are worse indicators in determining how similar an image is to a given one than tags such as *DigitalCreationDate* or *TimeSent*. Assessing the relevance of metadata tags is no easy task because image objects may have a variable number of them, depending on the device used to capture images or the format they are in. So, to ensure that a tag can be used in image similarity search, it has to appear in the images being compared and, at the same time, to feature a wide variety of values over the collection to prevent the overfitting of search results. In order to determine the most relevant tags for the task, an unsupervised learning algorithm was developed. It assigns a score to all the metadata tags of the images in the collection in which the search will be conducted. The score is a value between 0 and 1, where 0 is the lowest value in the uniqueness scale, meaning that the tag should not be considered at all, and 1 is the highest possible value, indicating that the tag is a perfect candidate.

Table 1
Structure of the dictionary containing the metadata tags for the calculation of uniqueness grades.

```
tags={
        "FlashFiring": [
        10000,
        set("Fires")],
        "ImageHash": [
        10000,
        set("aa91…fd1","ab12…91a", …, "9aaf…e21")]
        …
        "ColorTempFlash":[
        9783,
        set(20,22,15,…,18,70)]
        }
```

In a first stage, the algorithm creates a dictionary containing the name of the metadata tag as a *key* and, as a *value* for each *key*, an array with the number of times this metadata tag was used, along with all the different values for it. Table 1 shows a small example of this dictionary according to the already mentioned structure. The uniqueness grade of a tag is obtained by counting the number of different values a tag has and calculating the quotient of this number and the total number of occurrences of the tag. This procedure is done for each tag, using the information in the dictionary. Eq. (2) describes how this calculation is made. *uniquenessGrade*(*tag*) is the uniqueness grade for the tag, *differentOccurrences*(*tag*) are the number of different values found in the collection for *tag*, and *totalOccurrences*(*tag*) is the number of times the *tag* appears as a field in the collection.

$$uniquessGrade(tag) = \frac{differentOccurrences(tag)}{totalOcurrences(tag)} \quad (2)$$

The algorithm assigns to each tag its percentage of unique values. As an example, for the tag *FlashFiring* in Table 1 the uniqueness grade will be $1/10{,}000 \approx 0$. Therefore, the tag will not have a real impact in the characteristic curve of the image. On the other hand, if there is a unique hash for each document the grade will be $10{,}000/10{,}000 = 1$, which means that this tag will be really important in order to determine if an image is the one being searched. A tag such as *ColorTempFlash*, which only has a limited set of values (in this case 945 different values) and for which some images may have the same value, the uniqueness grade will be 945/9783=0.0945. This type of tag can help to determine if an image is similar to a given one, but not an exact match. This is a measure of pattern *frequency*. After applying this regularization to a sizable set of documents, like the collection used in this work, this information is stored in order to be used in the next stages of the algorithm.

3.2. *Retrieval of relevant tag groups*

In order to obtain a robust classification and determine how accurate the result of a search avoiding false positives is, it is necessary to generate a frequency curve for each group of tags that represent the metadata tags used in image retrieval. The frequency curve gathers information from all the tags belonging to a group in a single plot. As it will be discussed further in the paper, the curve denotes the average frequency of characters and the uniqueness of the tags used to build it. After the procedures in the previous section, it is now necessary to isolate groups containing the most common tags in the collection. Only then will it be possible to know which metadata tags should be used in the search. The best groups are those which contain the combinations of most frequent tags. In consecutive steps tags are added to the trie one at a time.

The Apriori algorithm is widely used in this type of problem and has been the object of intense study, which resulted in a great variety of implementations for the sake of improving its efficiency. This algorithm, or family of algorithms, use the Apriori property to reduce the search space. For its simplicity, speed and ease of understanding, this was also the approach followed to construct the groups with the most common tags. The variant of the algorithm chosen for the task was the *Depth First Implementation* [39]. In another work by [40] it is shown that this variant outperforms the FP-growth implementation in retail databases. Given the sample collection, this algorithm builds a trie in memory containing all the frequent groups of tags. Every path from the root of the trie downwards corresponds to a unique frequent group of tags. It requires four parameters as input: the minimum frequency in the collection to consider that a tag should be included in a group (*groupMinFreq*), which is to say the support of the tag, expressed as the ratio between the frequency of a tag in the items of a collection and the total number of items in the collection; the length of the groups in terms of number of tags (*groupLength*); and the minimum uniqueness grade of a tag for it to be included in a group (*groupMinUniqueness*). Since this is a depth first implementation, once a tag is selected by these criteria, it is combined with other existing tags to form groups with increasing cardinality. The groups to be kept have to obey the *groupMinFreq* restriction, meaning that the ratio between their occurrence as a group in the collection and the total number of items has to be higher or equal to this value too. The process unfolds until groups with the size of *groupLength* are retrieved, which will later be used for similarity search.

Additionally, the algorithm searches for the most important tags in the collection. So, in addition to the groups of size *groupLength* retrieved in the previous steps, groups containing only one tag are retrieved by selecting tags that have a minimum frequency higher or equal to *tagMinFreq* and a minimum uniqueness value higher or equal to *TagMinUniqueness*. Typically, the values of *tagMinFreq* and *tagMinUniqueness* should be higher than their counterparts, *groupMinFreq* and *groupMinUniqueness*, because the objective in this stage is to add to the set of already retrieved groups, the tags with most discriminating power out of the collection and assign them an importance in similarity search that is on pair with the other groups. To ensure that the system is able to deal with a query containing any tag, the last group added to the set of relevant groups contains all the different tags in the collection.

In sum, the process of extracting relevant groups from the object collection produces a set that contains groups of length *groupLength* obtained from the depth first implementation of Apriori, one tag groups representing the most important tags in the collection, and a group containing all possible tags. The parameters *groupMinFreq*, *groupLength*, *groupMinUniqueness*, *tagMinFreq*, and *TagMinUniqueness* are fully configurable, which allows the user to specify these values and tune the similarity search algorithm according to the desired performance.

3.3. *Representative curves for tag groups*

The objective of this stage is to obtain a representation of the items in the collection that can be used to compare them to another item used as a query. As stated above the metric space is one of such representations. This representation has to be numerical in order to support arithmetic operations. Moreover, the values must represent a distance. As such, each group of an item is represented by two vectors: *distVector* and *freqVector*.

For an item and for each of its groups, the algorithm retrieves the values of every tag of the group. Every value is processed as a string. The string is analyzed in order to determine the characters it contains. For each character found, its weight on the string is calculated according to Eq. (3), where *weight* (c,n) is the weight of character c within n positions from it in the string, $asciiValue(c)$ is the ASCII decimal value for character c and $asciiValue(c_i)$ is the ASCII decimal value of the character at position i starting from character c. With this the weight is sensible to the position of the character. If one takes the strings "star" and "arts" as examples, their characters will have different weights according to the position they are at for $n=3$. When comparing these strings to another one like "the star is bright", the weights of the characters in "star" will be closer in value than the characters in "arts", although they are the same, thus indicating that "star" is less distant from the query string. This way, it is possible to get better similarity results when comparing strings and substrings. If the characters are at different relative positions their distance increases.

$$weight(c,n) = \frac{\sum_{i=1}^{n} asciiValue(c)/asciiValue(c_i)}{n} \quad (3)$$

The distance value for a character c used in a group with m tags is obtained through Eq. (4). The multiplication by the uniqueness grade of tag_j aims at increasing the value of the distance measure when the character occurs in an important tag. The *distVector* is obtained by calculating *dist* for each occurring character.

$$dist(c,n,m) = \sum_{j=1}^{m} weight(c,n) \times uniqueness\ Grade(tag_j) \tag{4}$$

As for *freqVector* it is a frequency curve representing the average frequency of occurrence of each character in a group of tags of the image document. As it happens in the calculation of *distVector*, the strings for each tag in a group are calculated in order to determine the average frequency of their characters according to Eq. (5), where *avgFreq*(*c*, *str*) is the average frequency of character c in string *str*, *occurrences*(*c*) is the total number of occurrences of the character in the string, and *length*(*str*) is the total number of characters For instance, if the value of a tag is “aaeefs”, the frequency for the character a and e is 1/3, and for the characters f and s is 1/6. By multiplying the frequencies of each character by the uniqueness grade of the tag, and calculating the average of these values for all the characters across all tags of a group in an image, it is possible to plot a characteristic frequency curve per group. The calculation of the frequency values (*freq*(*c*,*str*,*m*)) for groups of m elements that will appear in *freqVector* is displayed in Eq. (6).

$$avgFreq(c,str) = \frac{accurences(c)}{length(str)} \tag{5}$$

$$freq(c,str,m) = \sum_{j=1}^{m} avgFreq(c,str) \times uniqueness\ Grade(tag_j) \tag{6}$$

The retrieved curves will represent a document. Using this method, two curves per group of tags will be calculated. To retain is that these curves will be used to find the most similar image document to a given one inside a set of documents, even if the image was modified.

The option believed to be the best to store all the curves was a MapReduce SQL like system, such as Hive [41]. Hive is an open source data warehouse solution built on top of Hadoop. It supports queries expressed in a SQL-like declarative language, which are compiled into map-reduce jobs that are executed using Hadoop. Hive and Hadoop are extensively used in Facebook for data processing. Hive provides a more powerful interface to Hadoop, facilitating the design of jobs and their implementation. In order to store all the curves for all the documents, a table is defined with a column to identify the document, another column to identify the group of the curve, and a column for the ASCII code of each character to be considered, as follows: *image_id*|*group_id*|32|33|34|…|124|125.

3.4. Search query

When an object is submitted as a search query, it goes through the same operations as the objects in the collection, which means calculating *distVector* and *freqVector* according to the equations described above. This is due to the need to reduce the object to the same form as the documents in the collection.

With the storage schema defined in the previous section, it is easy to build a SQL query to obtain a list of items sorted by proximity to a given one. Here, the concept of distance is defined as the sum of the differences between the *distVectors* and *freqVectors* of the search object and an object present in the collection. As such, the SQL query uses Eq. 7 to compare each object and each group in the collection with the search query. The element *distance*(*a*,*b*, *group*) stands for the distance between object a and object b, given *group*. As for *distVector*(*a*,*group*) and *distVector*(*b*, *group*), they are the *distVectors* of objects a and b respectively, for a certain *group* of tags. The same with *freqVector* (*a*,*group*) and *freqVector*(*b*,*group*). *numberDist* is the total number of distances calculated for each vector difference, which represents the number of characters present in both *distVector* and *freqVector*. This distance measure is a modified version of the Manhattan distance.

$$distance(a,b,group) = \frac{|distVector(a,group) - distVector(b,group)| + |freqVector(a,group) - freqVector(b,group)|}{number\ Dist} \tag{7}$$

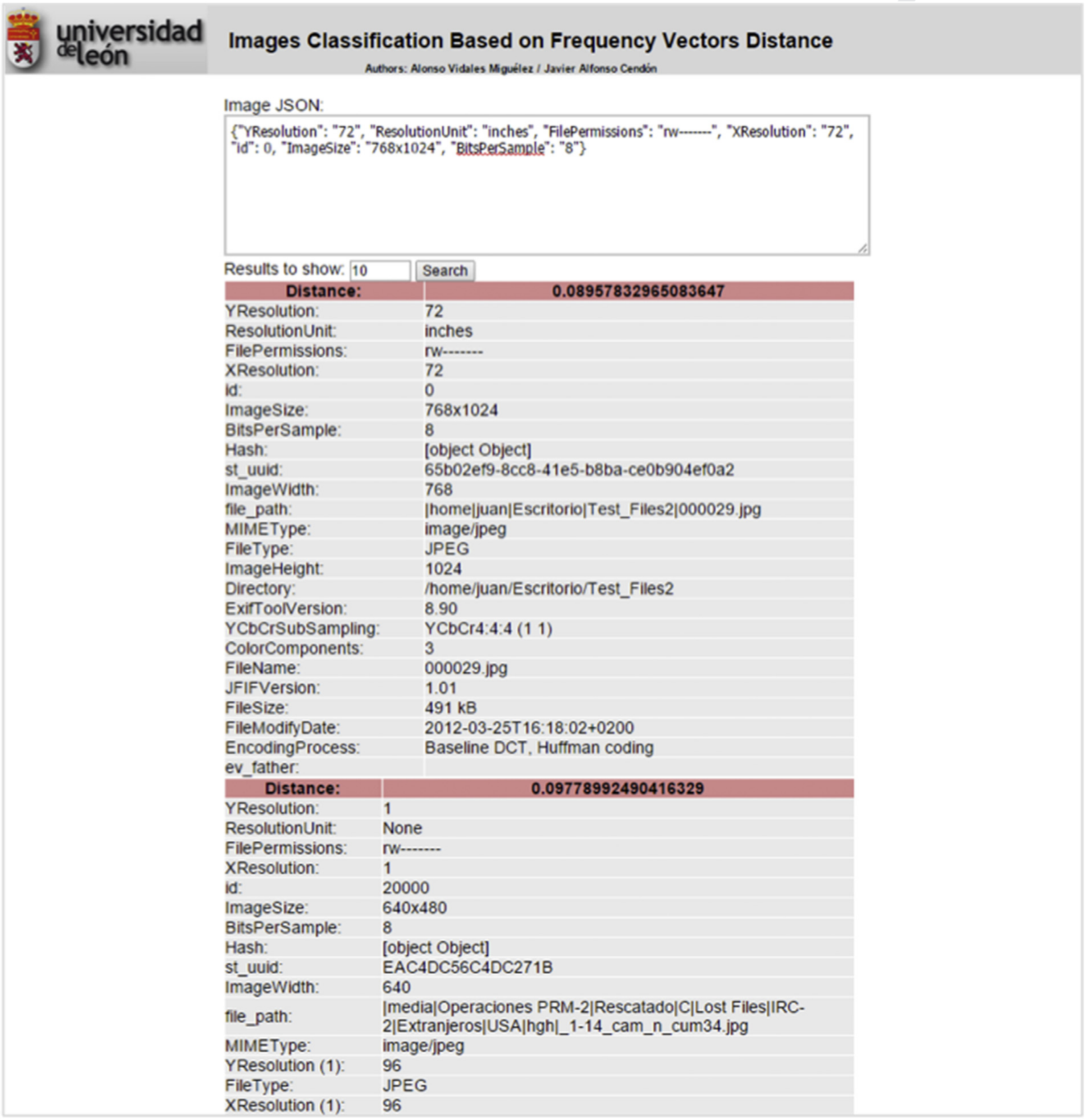
By doing this query for all the items inside each group of tags, and getting the k best matches, it is possible to mix all the results and obtain the average distance for all the curves, and the number of groups where the document was found within the first k matches. Computing the distance in each group of tags in parallel, it is basically a distribution of groups to be computed in parallel and a subsequent collect, obtaining the desired distance value. In addition to this, the distance between the object a and the object b, c or d can be carried out concurrently. After completing the search, it is possible to show a table sorted by average distance, and the number of groups used to calculate this distance. It is also possible with the reduce function, to obtain the average distances computed in parallel and merge them to obtain the minimum one.The search result can be sorted by the number of groups where the document was found within k matches, and the average distance to the given image. As such, the resulting list of documents includes the best possible matches.

The underlying idea is that, if the object to be searched is in the database, the distance between said object and the query object will be shorter when compared to any other object in the collection, even if the object was modified. If another image similar to the one used as a query exists in the collection, like, for instance, in the case of a picture took in the same location, on the same date, or with the same camera, the curve for these similar images will be closer than the rest of the images in the database. Thanks to that, it is possible to discover some useful relationships between groups of objects.

The system in which the algorithm is implemented has a web interface like the one shown in Fig. 1. In it, it is possible to specify the object for the query and the *k* closest results one wants to retrieve. The figure also demonstrates that it is possible to query with only a few metadata tags. In the example, one used a few metadata tags of an image in the collection with “id” = “0”. The top result is the image from which those tags were extracted.

3.5. Search parallelization

The algorithm was implemented following a MapReduce model [42] which is suitable for the processing and generation of large datasets. In it, users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all the intermediate values associated with the same intermediate key. Map invocations are distributed across multiple machines (or cores) by automatically partitioning the data in splits of equal size. The input splits can be processed in parallel by different machines.


universidad de león
Images Classification Based on Frequency Vectors Distance
Authors: Alonso Vidales Miguélez / Javier Alfonso Cendón
Image JSON:
Results to show:
Search

```
{"YResolution": "72", "ResolutionUnit": "inches", "FilePermissions": "rw-------", "XResolution": "72",
"id": 0, "ImageSize": "768x1024", "BitsPerSample": "8"}
```

Results to show: 10

Distance:	0.08957832965083647
YResolution:	72
ResolutionUnit:	inches
FilePermissions:	rw-------
XResolution:	72
id:	0
ImageSize:	768x1024
BitsPerSample:	8
Hash:	[object Object]
st_uuid:	65b02ef9-8cc8-41e5-b8ba-ce0b904ef0a2
ImageWidth:	768
file_path:	|home|juan|Escritorio|Test_Files2|000029.jpg
MIMEType:	image/jpeg
FileType:	JPEG
ImageHeight:	1024
Directory:	/home/juan/Escritorio/Test_Files2
ExifToolVersion:	8.90
YCbCrSubSampling:	YCbCr4:4:4 (1 1)
ColorComponents:	3
FileName:	000029.jpg
JFIFVersion:	1.01
FileSize:	491 kB
FileModifyDate:	2012-03-25T16:18:02+0200
EncodingProcess:	Baseline DCT, Huffman coding
ev_father:	

Distance:	0.09778992490416329
YResolution:	1
ResolutionUnit:	None
FilePermissions:	rw-------
XResolution:	1
id:	20000
ImageSize:	640x480
BitsPerSample:	8
Hash:	[object Object]
st_uuid:	EAC4DC56C4DC271B
ImageWidth:	640
file_path:	|media|Operaciones PRM-2|Rescatado|C|Lost Files|IRC-2|Extranjeros|USA|hgh|_1-14_cam_n_cum34.jpg
MIMEType:	image/jpeg
YResolution (1):	96
FileType:	JPEG
XResolution (1):	96

Fig. 1. The web interface for similarity search. It has an insertion point for the object query and another for the *k* results to retrieve.

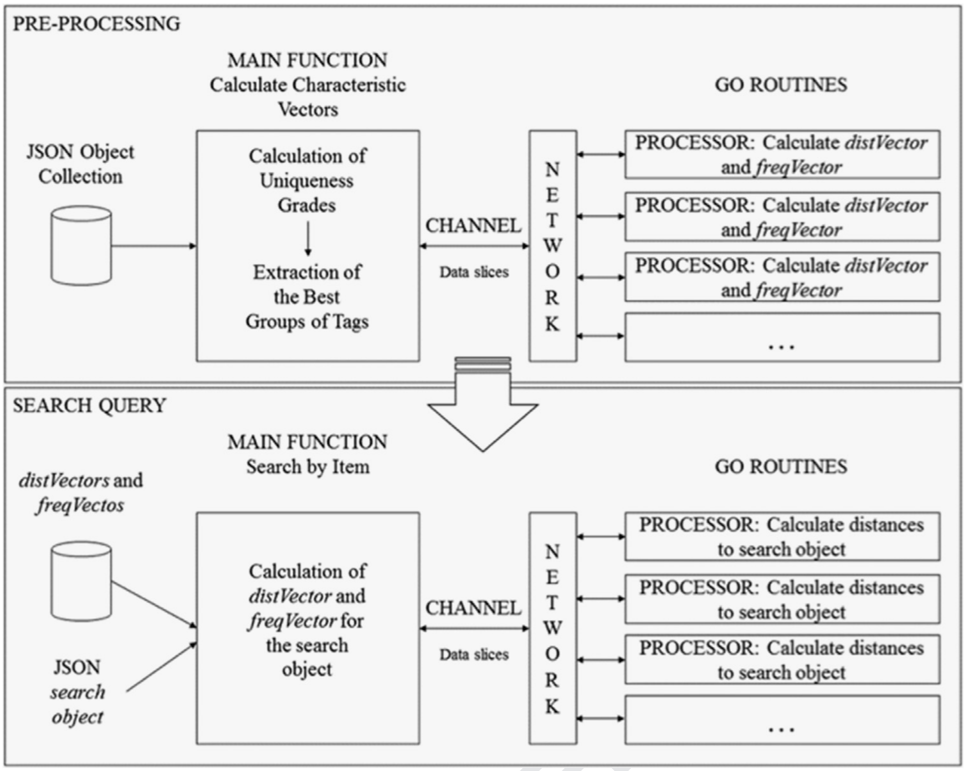

PRE-PROCESSING
MAIN FUNCTION
Calculate Characteristic Vectors
JSON Object Collection
Calculation of Uniqueness Grades
Extraction of the Best Groups of Tags
CHANNEL
Data slices
NETWORK
GO ROUTINES
PROCESSOR: Calculate distVector and freqVector
PROCESSOR: Calculate distVector and freqVector
PROCESSOR: Calculate distVector and freqVector
...
SEARCH QUERY
MAIN FUNCTION
Search by Item
distVectors and freqVectos
JSON search object
Calculation of distVector and freqVector for the search object
CHANNEL
Data slices
NETWORK
GO ROUTINES
PROCESSOR: Calculate distances to search object
PROCESSOR: Calculate distances to search object
PROCESSOR: Calculate distances to search object
...

Fig. 2. Diagram of the information flow as defined by the similarity search algorithm.

The main advantage of following this model is that programs are automatically parallelized and executed on a large cluster of machines. In [43] MacCreade et al. analyze the scalabity and efficiency of MapReduce, pointing out that multiple reduce tasks should be employed in order to obtain high parallelism and efficiency. In that work, a study of how the MapReduce indexing scales is presented, computing the same experiments for different corpus size and comparing the results. As in [43], here the textual terms are avoided and just considered distances and frequencies, so the scalability and efficiency is assured. Moreover, it is possible to fully channel the processing power of multicore machines to a given task or set of tasks. As stated above, the algorithm was developed using Go [8]. This programming language provides diverse concurrency patterns. These patterns are based on Go routines, which are independently executing functions, launched by Go statements. Each function has its own call stack. They are computationally cheaper than threads, and a running thread may have several Go routines. The communication between the main function and the boring functions is ensured by channels. This ensures that the code is able to handle multiple inputs and outputs and achieve synchronization between the running jobs.

The stages described above are parallelized according to this functional style. The whole process can be divided into two parts: Pre-processing and Search Query. These stages are depicted in Fig. 2. The main function in Pre-processing starts by calculating the uniqueness grades for all the tags in the collection. Afterwards, the best groups of tags for similarity search are extracted. Based on this information, the main function parallelizes the calculation of *distVector* and *freqVector* by splitting the image collection in equally-sized parts and distributing them by the processors (or cores) in the network. Each calculates the vectors for its slice of the data and sends the results back to the main function through a communication channel. In turn, the main function merges the outputs into a combined result. In Search Query, the main function has as the objective of retrieving the *k* items from the collection closest to the query object. To do that, first *distVector* and *freqVector* are calculated for the query object, then the main function parallelizes the calculation of the distances between this object and the items in the collection. The results are merged by the main function which presents the *k* closest matches.

The algorithm was tested on a collection consisting of 86,228 JSON metadata image records. The test was performed in a machine with an Intel® Core™ i3-2130 Processor (3M Cache, 3.40 GHz) with two cores and two threads per core. The input parameters for the algorithm were the ones specified in Table 2. The process of calculating the vectors for the items in the collection takes 2 min and 57 s, and it uses 100% of the cores in the machine. The query time with a random JSON object is placed between 5 and 6 s.

4. Discussion and results

4.1. Strengths of the approach

Given that this system looks for proximities in frequencies of characters rather than a perfect match, it becomes easier to detect proximities between items. For instance, a metadata tag of an image such as "createdDate", which contains the value "2013-10-01 11:12:32", and the same tag in another image took in the same session, have obviously a similar date value. In a case such as this, the system will detect the proximity between both items, and will be able to detect a relationship between them. This behavior may be useful in security in order to create a graph based on relationship between sex predators, terrorists, and so forth. Even when querying with incomplete information, using just a portion of the JSON object, the algorithm is able to find documents that bear a close resemblance to the query.

Table 2
Parameters for the similarity search algorithm.

groupMinFreq	0.2
groupLength	7
groupMinUniqueness	0.5
tagMinFreq	0.8
tagMinUniqueness	0.8
n	4

Table 3
Parameters for the similarity search algorithm.

```
hash = ''
for cell in image_cells:
    pixel_xor = 0
    for pixel in cell:
        for color in pixel:
            pixel_xor ^= color
    hash += pixel_xor
        hash = base64(hash)
```

The approach followed herein does not have any parallel in the current state of the art, and its main point is to take advantage of existing frameworks and resources for parallelization in order to build a similarity search system that does not require a complex processing of a document collection. In fact, using this approach removes the need to perform text mining operations or to have complex indexing schemes to save computational resources. Indeed, it becomes unnecessary to resort to partitioning strategies and index structures such as the Burkhard-Keller Tree, the Vantage Point Tree, the Bisector Tree, or other variations mentioned in the related work section. By avoiding this, it is also possible to avoid the performance bottlenecks that are usually associated with these structures. Adding new elements to the search collection also becomes quite simple since it is only necessary to add a record to the Hive database. It is not necessary to update a search structure in the process. Moreover, by using character frequencies, this search method can be applied to different domains, without the need for further alterations. As such, it can be easily applied to metadata of other types of document. Another advantage is that it is not necessary to know beforehand which metadata tags describe and image, nor their number. In truth, the items of the collection may differ in the number of tags that describe them.

This method for classification and search on images can be combined with hash generation methods based on the content of the items. It is possible to analyze the different elements in the content of a document. For instance, in the case of a text document, one can store a hash with the

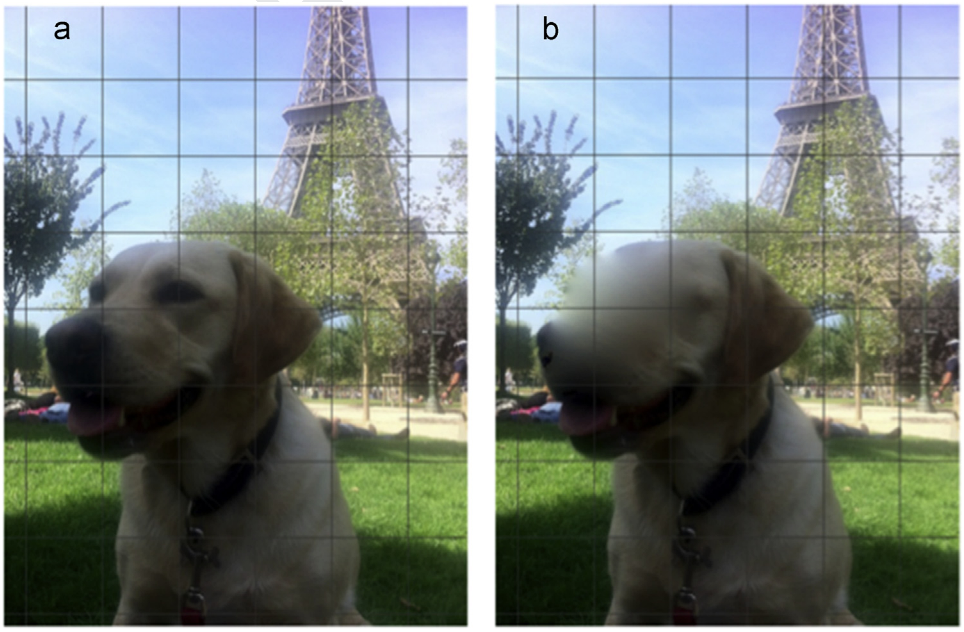


Fig. 3. (a) Unmodified image and (b) version of the same image modified in six of its 48 cells.

frequencies for each character dividing the document in shards. In doing so, if a part of the content is modified, one can still get a hash closest to the original one. For an image object, in order to obtain the most similar hash even if the image was modified, one may split the image in cells and perform a "xor" operation for all the color values in each pixel for each cell according to the code in Table 3.

Considering two images such as the ones shown in Fig. 3, in which (a) is the original image and (b) is a version of it modified in six of the forty-eight cells of the grid. Then, if the hash of the original is: "adc6888dd489ff0514f1102ad48bd90-badc6888dd489". For the retouched image one gets a hash like "adc6888dd489ff0514f1102eaa1c990badc6888dd489". The algorithm will be able to detect the proximity between two images even if the images were modified, thanks to the similarity of the hashes.

4.2. Limitations of the approach

The characteristic curves produced by the algorithm result from merging distance and frequency values of different tags belonging to a group into a concise representation. When doing this, it becomes possible to perform approximate similarity search, but, at the same time, it becomes nearly impossible to get distances to the object query of value 0. The top result will, at most, have a value very close, which means that it is very difficult to determine, with absolute certainty, if an image is an exact match. However, this aspect is compensated by the ability of the algorithm to find relationships between items. This is a trade-off that, depending on the application, may produce results that fall short of the objective. The algorithm may produce false positives if the items in the collection do not have enough information to classify them. If the items do not have enough tags to cross information, an item may be retrieved just because of the value of one tag. If that tag does not have a high uniqueness grade, which depends mostly on the initial parameterization of the algorithm.

Another problem is the complexity. This method needs a full scan of the documents stored in the database in order to determine the distance to the object query. As such, algorithmic complexity is $O(n^*m)$ where n is the number of documents stored on the database and m is the number of curves per document. This computational complexity is higher than that of the approaches mentioned in Section 2.1 [9]. However, the algorithm was specifically developed having parallelization in mind. The comparisons between documents is fairly fast, but in case of a really big database (more than a billion of documents, for instance), it may become difficult to handle. When there is a need for heavy usage of this system, Hive provides a good solution for the search process.

5. Conclusions and future work

The great deployment of computational resources experienced in recent years, calls forth the need for solutions which fully exploit the computational power of the available technology. The central concepts to this are scalability, performance, and customization. These are the features that a similarity search framework should provide. To achieve this, newly developed search mechanisms should be general-purpose and highly extensible. Similarity between objects is very subjective, thus very difficult to express by a unique rigorous function.

In light of this, the major contributions of this work are the following. Firstly, it provides a similarity search algorithm that can be applied to images, independently of their content. It is solely based on metadata and it is also able to deal with heterogeneous collections in terms of the number and variety of tags that their items have. The algorithm provides a method for extracting the most relevant metadata tag groups from collections containing image objects in different formats and was designed to be implemented with a MapReduce strategy. Another major contribution is the representation of items in the metric space using the frequencies of characters in tag values. The algorithm is capable of performing $kNN(q)$ queries by approximation to these frequency curves and is ideal for establishing origin relationships between objects. Furthermore, this strategy can be generalized to other types of documents besides images.

As future work, it is important to tackle the issues of false positive retrieval and computational complexity. Solutions may lie in testing other types of representations for the objects in combination with the frequency curves and including more elaborate data structures in order to increase performance and reduce complexity.

Acknowledgments

This work has been done in the context of the project "ASASEC (Advisory System Against Sexual Exploitation of Children)" (HOME/2010/ISEC/AG/043) supported by the European Union with the program "Prevention and fight against crime".

References

[1] J. Gantz, D. Reinsel, The Digital Universe in 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East, International Data Corporation, 2012.
[2] G. Ritzer, N. Jurgenson, Production, consumption, prosumption: the nature of capitalism in the age of the digital 'prosumer', J. Consum. Cult. 10 (2010) 13–36.
[3] R. Cattell, Scalable SQL and NoSQL data stores, SIGMOD Rec. 39 (2011) 12–27.
[4] C. Bohm, S. Berchtold, D.A. Keim, Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases, ACM Comput. Surv. 33 (2001) 322–373.
[5] M. Patella, P. Ciaccia, Approximate similarity search: a multi-faceted problem, J. Discret. Algorithms 7 (2009) 36–48.
[6] J. Tesic, Metadata practices for consumer photos, Multimed., IEEE 12 (2005) 86–92.
[7] J. Pokorny, NoSQL databases: a step to database scalability in web environment, Int. J. Web Inf. Syst. 9 (2013) 69–82.
[8] Doxsey, C., 2012. An Introduction to Programming in Go.
[9] P. Zezula, G. Amato, V. Dohnal, M. Batko, Similarity Search: The Metric Space Approach, Springer, 2006.
[10] E. Chavez, G. Navarro, R. Baeza-Yates, J. Marroquin, Searching in metric spaces, ACM Comput. Surv. 33 (2001) 273–321.

[11] G.R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces (Survey Article), ACM Trans. Database Syst. 28 (2003) 517–580.
[12] J.B. Kruskal, Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis, Psychometrika 29 (1964) 1–27.
[13] J. Uhlmann, Implementing metric trees to satisfy general proximity/similarity queries, in: Proceedings of the Command Control Symposium, Washington, DC, 1991.
[14] P.N. Yianilos, Excluded middle vantage point forests for nearest neighbor search, in: DIMACS Implementation Challenge, ALENEX'99, Citeseer, 1999.
[15] W.A. Burkhard, R.M. Keller, Some approaches to best-match file searching, Commun. ACM 16 (1973) 230–236.
[16] P.N. Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, in: Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 1993, pp. 311–321.
[17] Bozkaya, T., Ozsoyoglu, M., 1997. Distance-based indexing for high-dimensional metric spaces, Proceedings of the 1997 ACM SIGMOD international conference on Management of data. ACM, Tucson, Arizona, USA, pp. 357–368.
[18] S. Berchtold, C. Bohm, H.-P. Kriegal, The pyramid-technique: towards breaking the curse of dimensionality, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, ACM, Seattle, Washington, USA, 1998, pp. 142 – 153.
[19] G.R. Hjaltason, H. Samet, Ranking in Spatial Databases, Advances in Spatial Databases, Springer, 1995, 83–95.
[20] I. Kalantari, G. McDonald, A data structure and an algorithm for the nearest point problem, IEEE Trans. SE-Software Engineering 9 (1983) 631–634.
[21] F. Dehne, H. Noltemeier, Voronoi trees and clustering problems, in: G. Ferraté, T. Pavlidis, A. Sanfeliu, H. Bunke (Eds.), Syntactic and Structural Pattern Recognition, Springer Berlin Heidelberg, 1988, pp. 185–194.
[22] P. Ciaccia, M. Patella, P. Zezula, M-tree: an efficient access method for similarity search in metric spaces, in: Proceedings of the 23rd International Conference on Very Large Data Bases. Morgan Kaufmann Publishers Inc., 1997, pp. 426-435.
[23] T. Skopal, J. Pokorný, M. Krátký, V. Snášel, Revisiting M-tree building principles, in: L. Kalinichenko, R. Manthey, B. Thalheim, U. Wloka (Eds.), Advances in Databases and Information Systems, Springer Berlin Heidelberg, 2003, pp. 148–162.
[24] J. Caetano Traina, A.J.M. Traina, B. Seeger, C. Faloutsos, Slim-Trees: high performance metric trees minimizing overlap between nodes, in: Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology, Springer-Verlag, 2000, pp. 51–65.
[25] R. Agrawal, T. Imieliński, A. Swami, Mining association rules between sets of items in large databases, ACM SIGMOD Rec. (1993) 207–216.
[26] R. Agrawal, R. Srikant, R., Fast algorithms for mining association rules, in: Proceedings of the 20th International Conference on Very Large Data Bases, VLDB, 1994, pp. 487–499.
[27] M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, Parallel algorithms for discovery of association rules, Data Min. Knowl. Discov. 1 (1997) 343–373.
[28] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, SIGMOD Rec. 29 (2000) 1–12.
[29] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data Min. Knowl. Discov. 15 (2007) 55–86.
[30] R. Agrawal, J.C. Shafer, Parallel mining of association rules, IEEE Trans. Knowl. Data Eng. 8 (1996) 962–969.
[31] K.P. Kumar, S. Arumugaperumal, An analytical study on frequent itemset mining algorithms, in: R. Prasath, T. Kathirvalavakumar (Eds.), Mining Intelligence and Knowledge Exploration, Springer International Publishing, 2013, pp. 611–617.
[32] B. Fernando, E. Fromont, T. Tuytelaars, Effective use of frequent itemset mining for image classification, in: A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, C. Schmid (Eds.), Computer Vision–ECCV 2012, Springer Berlin Heidelberg, 2012, pp. 214–227.
[33] H. Cheng, X. Yan, J. Han, C.-W. Hsu, Discriminative frequent pattern analysis for effective classification, Data Engineering, in: Proceedings of the IEEE 23rd International Conference on ICDE, 2007, pp. 716–725.
[34] NISO, Understanding metadata, Natl. Inf. Stand. Organ. (2004).
[35] MWG, Guidelines for hanging metadata, Metadata Work. Group (2010).
[36] K.-P. Yee, K. Swearingen, K. Li, M. Hearst, Faceted metadata for image search and browsing, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, Ft. Lauderdale, Florida, USA, 2003, pp. 401–408.
[37] D. Korenblum, D. Rubin, S. Napel, C. Rodriguez, C. Beaulieu, Managing biomedical image metadata for search and retrieval of similar images, J. Digit. Imag. 24 (2011) 739–748.
[38] L. Zhang, L. Chen, F. Jing, K. Deng, W.-Y. Ma, EnjoyPhoto: a vertical image search engine for enjoying high-quality photos, in: Proceedings of the 14th Annual ACM International Conference on Multimedia, ACM, Santa Barbara, CA, USA, 2006, pp. 367–376.
[39] W. Kosters, W. Pijls, V. Popova, Complexity analysis of depth first and FP-growth implementations of APRIORI, in: P. Perner, A. Rosenfeld (Eds.), Machine Learning and Data Mining in Pattern Recognition, Springer Berlin Heidelberg, 2003, pp. 284–292.
[40] W.A. Kosters, W. Pijls, Apriori, A Depth First Implementation, in: Proceedings of the Workshop on Frequent Itemset Mining Implementations, 2003.
[41] D. Borthakur, J. Gray, J.S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, S. Rash, Apache Hadoop goes realtime at Facebook, in: Proceedings of the 2011 ACM SIGMOD International Conference on Management of data, ACM, 2011, pp. 1071-1080.
[42] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113.
[43] R. McCreadie, et al., MapReduce indexing strategies: studying scalability and efficiency, Inf. Process. Manag. http://dx.doi.org/10.1016/j.ipm.2010.12.003.