RankReduce – Processing K-Nearest Neighbor Queries on Top of MapReduce[*]

Aleksandar Stupar
Saarland University
Saarbrücken, Germany
astupar@mmci.uni-saarland.de

Sebastian Michel
Saarland University
Saarbrücken, Germany
smichel@mmci.uni-saarland.de

Ralf Schenkel
Saarland University
Saarbrücken, Germany
schenkel@mmci.uni-saarland.de

ABSTRACT

We consider the problem of processing K-Nearest Neighbor (KNN) queries over large datasets where the index is jointly maintained by a set of machines in a computing cluster. The proposed RankReduce approach uses locality sensitive hashing (LSH) together with a MapReduce implementation, which by design is a perfect match as the hashing principle of LSH can be smoothly integrated in the mapping phase of MapReduce. The LSH algorithm assigns similar objects to the same fragments in the distributed file system which enables a effective selection of potential candidate neighbors which get then reduced to the set of K-Nearest Neighbors. We address problems arising due to the different characteristics of MapReduce and LSH to achieve an efficient search process on the one hand and high LSH accuracy on the other hand. We discuss several pitfalls and detailed descriptions on how to circumvent these. We evaluate RankReduce using both synthetic data and a dataset obtained from Flickr.com demonstrating the suitability of the approach.

1. INTRODUCTION

With the success of the Web 2.0 and the wide spread usage of cell phones and digital cameras, millions of pictures are being taken and uploaded to portals like Facebook or Flickr every day[1], accumulating to billions of images[2]. Searching in these huge amounts of images becomes a challenging task. While there is an increasing trend to use social annotations, so-called tags, for image retrieval, next to the traditional image search à la Google/Bing/Yahoo which inspects the text around the web site holding the picture, there is a vital need to process similarity queries, where for a given query picture, the K most similar pictures are returned based on low level features such as color, texture, and shape [9]. The big advantage of such low level features is that they are always available, whereas tags are usually extremely scarce and text around images can often be misleading. Approaches like the work by Taneva et al. [24] use both textual and low level image descriptors to increase the diversity of returned query results. There exist plenty of fundamental prior works [16, 5, 4] on how to index feature based representations of pictures or, more generally, high dimensional vectors in a way that allows for inspecting only a small subset of all vectors to find the most similar ones.

The increasing volume of high dimensional data, however, poses novel problems to traditional indexing mechanisms which usually assume an in-memory index or optimize for local disk access. As a promising approach to process huge amounts of data on a multitude of machines in a cluster, MapReduce [10] has been proposed and continuously explored for many interesting application classes. In this paper, we investigate the usage of MapReduce for searching in high dimensional data. Although MapReduce was initially described in a generic and rather imprecise way in terms of implementation, implementations like Apache's Hadoop have proven to provide salient properties such as scalability, ease of use, and most notably robustness to node failures. This provides an excellent base to explore MapReduce for its suitability for large scale management of high dimensional data.

In this work, we propose RankReduce, an approach to implement locality sensitive hashing (LSH) [1, 8, 16], an established method for similarity search on high dimensional data, on top of the highly reliable and scalable MapReduce infrastructure. While this may seem to be straight forward at first glance, it poses interesting challenges to the integration: most of the time, we face different characteristics of MapReduce and LSH which need to be harnessed both at the same time to achieve both high accuracy and good performance. As MapReduce is usually used only to process large amounts of data in an offline fashion and not for query processing, we carefully investigate its suitability to handle user defined queries effectively demonstrating interesting insights on how to tune LSH on top of MapReduce.

The remainder of the paper is structured as follows. Section 2 gives an overview of related work, Section 3 presents our framework and gives a brief introduction to LSH and MapReduce, Section 4 describes the way queries are processed, Section 5 presents an experimental evaluation, and Section 6 concludes the paper and gives an outlook on ongoing work.

*This work has been supported by the Cluster of Excellence "Multimodal Computing and Interaction" (MMCI)

[1]http://blog.facebook.com/blog.php?post=2406207130

[2]http://blog.flickr.net/en/2009/10/12/4000000000/

2. RELATED WORK

Processing K-Nearest Neighbor queries in high dimensional data has received a lot of attention by researchers in recent years. When the dimensionality increases the distance between the closest and the farthest neighbor decreases rapidly, for most of the datasets [6], which is also known as the 'curse of dimensionality'. This problem has a direct impact on exact KNN queries processing based on tree structures, such as X-Tree [5] and K-D tree [4], rendering these approaches applicable only to a rather small number of dimensions. A better suitable approach for KNN processing in high dimensions is Locality Sensitive Hashing (LSH) [1, 8, 16]. It is based on the application of locality preserving hash functions which map, with high probability, close points from the high dimensional space to the same hash value (i.e., hash bucket). Being an approximate method, the performance of LSH highly depends on accurate parameter tuning [12, 3]. Work has also been done on decreasing the number of hash tables used for LSH, while preserving the precision, by probing multiple buckets per hash table [20]. In recent years, a number of distributed solutions where the main emphasis was put on exploring loosely coupled distributed systems in form of Peer-to-Peer networks (P2P) such as [11, 13, 14, 17, 23]) have been proposed, cf. the work by Batko et al. [2] for a discussion on the suitablity of different P2P approaches to distributed similarity search.

MapReduce is a framework for efficient and fault tolerant workload distribution in large clusters [10]. The motivation behind the design and development of MapReduce has been found in Information Retrieval with its many computationally expensive, but embarrassingly parallel problems on large datasets. One of the most basic of those problems is the inverted index construction, described in [21]. MapReduce has not yet been utilized for distributed processing of KNN queries. Some similarities with KNN processing can be found in recent work by Rares et al. [22] which describes a couple of approaches for computing set similarities on textual documents, but it does not address the issue of KNN query processing. The pairwise similarity is calculated only for documents with the same prefixes (prefix filtering), which can be considered as the LSH min-hashing technique. Lin [19] describes a MapReduce based implementation of pairwise similarity comparisons of text documents based on an inverted index.

3. RANKREDUCE FRAMEWORK

We address the problem of processing K-Nearest Neighbor queries in large datasets by implementing a distributed LSH based index within the MapReduce Framework.

An LSH based index uses *locality sensitive hash* functions for indexing data. The salient property of these functions is that they map, with high probability, similar objects (represented in the d-dimensional vector space) to the same hash bucket, i.e., related objects are more probable to have the same hash value than distant ones. The actual indexing builds several hash tables with different LSH functions to increase the probability of collision for close points. At query time, the KNN search is performed by hashing the query point to one bucket per hash table and then to rank all discovered objects in any of these buckets by their distance to the query point. The closest K points are returned as the final result.

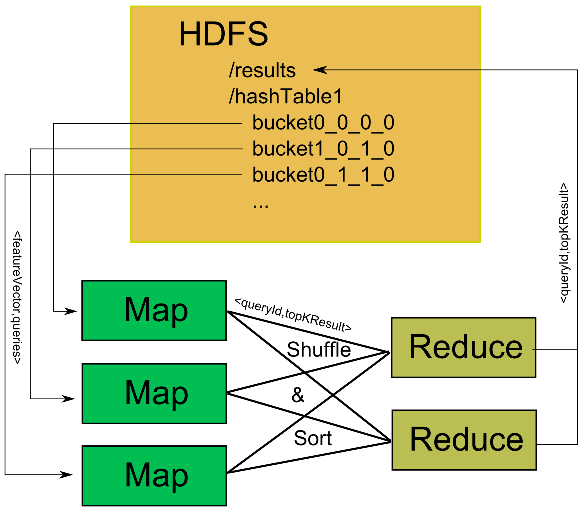

Figure 1: The RankReduce Framework

In this work, we consider the family of LSH functions based on p-stable distributions [8] which are most suitable for l_p norms. In this case, for each data point $\mathbf{v}$, the hashing scheme considers k independent hash functions of the form

$$h_{\mathbf{a},B}(\mathbf{v}) = \lfloor \frac{\mathbf{a}\cdot\mathbf{v}+B}{W} \rfloor \qquad (1)$$

where $\mathbf{a}$ is a d-dimensional vector whose elements are chosen independently from a p-stable distribution, $W \in \mathbb{R}$, and B is chosen uniformly from $[0, W]$. Each hash function maps a d-dimensional data point onto the set of integers. With k such hash functions, the final result is a vector of length k of the form $g(\mathbf{v}) = (h_{\mathbf{a_1},B_1}(\mathbf{v}), ..., h_{\mathbf{a_k},B_k}(\mathbf{v}))$.

In order to achieve high search accuracy, multiple hash tables need to be constructed. The work by Lv et al. [20] presents an approach to probe multiple buckets per hash table which, however, leads to either sampling a larger fraction of the dataset or to many fine grained accesses to small buckets. The latter causes a larger number of expensive random accesses to the underlying infrastructure, as we deal with file based indexes as opposed to in-memory accesses. Hence, we opted for using a single probe per hash table.

For maintaining the hash tables over a set of machines in a cluster, we employ MapReduce [10] which is designed to be used for large data processing in parallel. It is built on top of the Distributed File System [15], which enables distributing the data over the cluster machines in a scalable and fault tolerant way. This tight integration of MapReduce with the distributed file system enables it to move calculations where the data resides, eliminating network bandwidth bottlenecks caused by data shipping during query processing. Our implementation uses the open source software Hadoop [3], maintained by the Apache Foundation, which provides a Java based implementation of both the MapReduce framework and the Distributed File System (coined HDFS for Hadoop Distributed File System). In the last years, Hadoop gained a lot of popularity in the open source community and is also part of many research efforts investigating large data processing.

[3]http://hadoop.apache.org/

MapReduce is a fairly simple programming model, based on two developer supplied functions: *Map* and *Reduce*. Both functions are based on key-value pairs. The Map function receives a key-value pair as input and emits multiple (or none) key-value pairs as output. The output from all Map functions is grouped by key, and for each such key, all values are fed to the Reduce function, which then produces the final output from these values.

In the Hadoop implementation, the input data is grouped in so-called *input splits* (which often correspond to blocks in the distributed file system), and a number of so-called *mapper* processes call the Map function for each key-value pair in such an input split. A number of mappers can run concurrently on each node in the cluster, and the mapper processes are in addition distributed over all nodes in the cluster. Ideally, a mapper is run on the same node where the input block resides, but this is not always possible due to workload imbalance. Similarly, after all mappers have finished, dedicated *reducer* processes are run on nodes int the cluster. Each reducer handles a fraction of the output key space, copies those key-value pairs from all mappers' outputs (in the so-called *shuffle phase*), sorts them by key, and feeds the to the Reduce function. The output of the reducers is usually considered the final result but can also be used as input for following MapReduce jobs.

Figure 1 shows an illustration of our LSH integration with MapReduce. Each hash table in the LSH index is mapped to one folder in HDFS. For each bucket in such a hash table, a corresponding file is created in this folder, where the file name is created by concatenating hash values into a string, with '_' as separator. This mapping of buckets to HDFS files enables fast lookup at query time and ensures that only data that is to be probed is read from the HDFS. Placing the bucket in one file also enables block based sequential access to all vectors in one bucket, which is very important as the MapReduce framework is optimized for such block based rather than random access processing. Each of the buckets stores the complete feature vectors of all objects mapped to this bucket in a binary encoding.

Indexing of new feature vectors to the LSH index in HDFS is easily done by appending them to the end of the appropriate bucket file. This can also be done in parallel with query processing as long as different buckets are affected; as HDFS does not include a transaction mechanism, appending entries to buckets that are being queried would be possible, but with unclear semantics for running queries. As HDFS scales well with increasing cluster size, the resulting growth of the LSH index can easily be supported by adding more machines to the cluster.

While an LSH index stored in-memory has no limitation on the number of buckets, too many files in HDFS can downgrade its performance, especially if these files are much smaller than the block size (which defaults to 64MB). The number of buckets, and therefore the number of files in HDFS for the LSH index, is highly dependent on the set up of LSH parameters as choosing a bad combination of parameters can result in a large number of small files.

Inspired by in-memory indexes which can have references from buckets to materialized feature vectors, we considered storing only feature vector ids in the buckets instead of the actual feature vectors, and retrieving the full vectors only on demand at query time. However, this approach would result in poor performance due to many random accesses to the HDFS when retrieving the full vectors, so we decided to store complete feature vectors. This fact also needs to be addressed when setting up LSH parameters, while too many LSH hash tables can dramatically increase index size, as each feature vector is materialized for each hash table.

4. QUERY PROCESSING

We implemented KNN query processing as a MapReduce job. Before starting this MapReduce job, the hash values for the query documents are calculated. These values are then used for selecting the buckets from the LSH index, which are to be probed. The selected buckets are provided as input to the query processing MapReduce job, generating multiple input splits. The generated input splits are read by a custom implementation of the *InputFormat* class, which reads feature vectors stored in a binary format and provides them as the key part of the Map function input. Queries are being distributed to mappers either by putting them in the *Distributed Cache* or by putting them in HDFS file with high number of replicas. They are read once by the InputFormat implementation and reused as value part of the Map function input between the function invocations.

The input to the Map function consists therefore of the feature vector to be probed as the key and the list of queries as the value. The Map function computes the similarity of the feature vector with all query vectors. While a standard MapReduce implementation would now emit a result pair for each combination of feature vector and query vector, we employ an optimization that delays emitting results until all feature vectors in the input split have been processed. We then eventually emit the final K-Nearest Neighbor for each query vector from this input split in the form of key-value pairs. Here, the query is the key and a nearest neighbor together with its distance to the query vector is the value. To implement this delayed emitting, we store the currently best K-Nearest Neighbor for each query in-memory, together with their distances from the query points. The results are emitted at the end of processing the input split in Hadoop's cleanup method[4]. The Reduce method then reads, for each query, the K-Nearest Neighbor from each mapper, sorts them by increasing distance, and emits the best K of them as the final result for this query.

The final sort in the reducer can even be executed within Hadoop instead of inside the Reduce method, as a subtask of sorting keys in the reducer. It is possible to apply a so-called *Secondary Sort* that allows, in our application, to sort not just the keys, but also the values for the same key. Technically, this is implemented by replacing, for each (query, (neighbor, distance)) tuple that is emitted by a mapper, the key by a combined key consisting of the query and the distance. Keys are then sorted lexicographically first by query and then by distance. For assigning tuples to a Reduce method, however, only the query part of the key is taken into account. The reducer then only needs to read the first K values for each key, which then correspond to the K-Nearest Neighbor for that query.

It is worth mentioning that because one feature vector is placed in multiple hash tables, the same vector can be evaluated twice for the same query during processing. An

[4]This feature was introduced in the most recent version 0.20; before, it was only possible to emit directly from the Map function

alternative approach would be to have two MapReduce jobs for query processing instead of one, which would eliminate this kind of redundancy. The first MapReduce job would create a union between buckets that need to be probed, and the second job would use the union as an input to similarity search. However, while this would possibly save redundant computations, it has the major drawback that the results from the first job need to be written to the HDFS before starting the second job. As the overhead from multiple evaluations of the same feature vector has not been too large in our experimental evaluation (see Figure 4), we decided that it is better to probe slightly more data rather than to pay the additional I/O cost incurred by using two Map Reduce jobs.

The approach can handle multiple queries at the same time in one MapReduce job. But it is not suitable for the cases when the number of queries becomes too large, as problem of KNN queries processing becomes the problem of set similarity joins [22].

5. EXPERIMENTAL EVALUATION

For our experiments we have used Hadoop version 0.20.2 installed on three virtual machines with Debian GNU/Linux 5.0 (Kernel version: 2.6.30.10.1) as operating system. Each of the virtual machines has been configured to have 200GB hard drive, 5 GB main memory and two processors. VMware Server version 2.0.2 was used for virtualization of all machines. The virtual machines were run on a single machine with Intel Xeon CPU E5530 @2.4 GHz, 48 GB main memory, 4 TB of hard drive and Microsoft Windows Server 2008 R2 x64 as operating system. We used a single machine Hadoop installation on these virtual machines as described later on.

Datasets

As the performance of the LSH based index is highly dependent on the data characteristics [12], we conducted an experimental evaluation both on randomly generated (Synthetic Dataset) and real world image data (Flickr Dataset):

Synthetic Dataset:
For the synthetic dataset we used 32-dimensional randomly generated vectors. The synthetic dataset was built by first creating N independently generated vector instances drawn from the normal distribution $N(0, 1)$ (independently for each dimension). Subsequently, we created m near duplicates for each of the N vectors, leading to an overall dataset size of $m*N$ vectors. The rational behind using the near duplicates is that we make sure that the KNN retrieval is meaninful at all. We set m to 10 in the experiments and adapt N to the desired dataset size depending on the experiment. We generated 50 queries by using the same procedure as the original vectors were generated.

Flickr Dataset:
We used the 64-dimensional color structure feature vectors from crawled Flickr images provided by the CoPhIR data collection [7] as our real image dataset. We extracted the color structure feature vectors from the available MPEG-7 features and stored them in a binary format suitable for the experiments. As the queries, we have randomly selected 50 images from the rest of the CoPhIR data collection

As LSH is an approximate method, we measure the effectiveness of the nearest neighbor search by its *precision*, which is the relative overlap of the true K-Nearest Neighbor with the K-Nearest Neighbor computed by our method. And for the proximity measure we used Euclidean distance.

5.1 LSH Setup

Before starting the evaluation we needed to understand how to set up LSH and what consequence it may have on the index size and query performance. In our setup we consider the number of hash tables and the bucket size as LSH parameters to be tuned. The bucket size can be changed either by changing the number of concatenated hash values or by changing the parameter W in Formula 1. Because W is a continuous variable and provides a subtle control over bucket size, we first fix the number of concatenated hash values and then vary parameter W [12]. These two parameters together determine which subset of the data needs to be accessed to answer a query (one bucket per hash table). We varied the bucket size by varying parameter W for a different number of hash tables and then measured data subset probed and precision, shown in Figure 2 for synthetic dataset and in Figure 3 for the Flickr dataset. These measurements were done using 50 KNN queries for $k = 20$ on both datasets, but with reduced sizes to 100,000 feature vectors indexed. The results show that increasing the number of hash tables can decrease the data subset that needs to be probed to achieve a certain precision, resulting in less time needed for the query execution.

Realizing that each new table creates another copy of data and we may have only limited storage available, we need to tradeoff storage cost vs. execution time. Additionally, when only a fixed subset of the data should be accessed, a larger number of hash tables results in a large number of small sized buckets, which is not a good scenario for HDFS (it puts additional pressure on Hadoop's data node that manages all files). On one hand, we would like to increase the number of hash tables and to decrease the probed data subset. On the other hand, we would like to use less storage space and a smaller number of files for storage and probing. Figure 3 shows that the number of hash tables has smaller impact on precision in case of real image data. Thus, as a general rule we suggest a smaller number of hash tables with larger bucket sizes, still set to satisfy the precision threshold. Therefore we settle for a setup of four hash tables and a bucket size that allow us to get at least 70% precision.

5.2 Evaluation

We evaluate our approach and compare it to the linear scan over all data, also implemented as a MapReduce job. As we did not have a real compute cluster at hand for running the experiments, we simulate the execution in a large cluster by running the mappers and reducers sequentially on our small machine. We measure run times of their executions and the number of mappers started for each query job. To avoid the possible bottleneck of a shared hard drive between virtual machines [18], we run each experiment on a single machine Hadoop installation with one map task allowed per task tracker. This results in sequential execution of map tasks so there is no concurrent access to a shared hard drive by multiple virtual machines.

Considering that the workload for the reducers is really small for both linear scan and LSH, we only evaluate map execution times and the number of mappers run per query job. We measured the map execution times for all jobs and found that they are approximately constant, with average value per mapper being 3.256 seconds and standard devia-

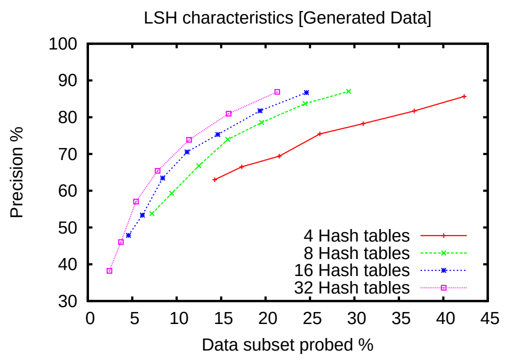

Figure 2: LSH characteristics on generated data.

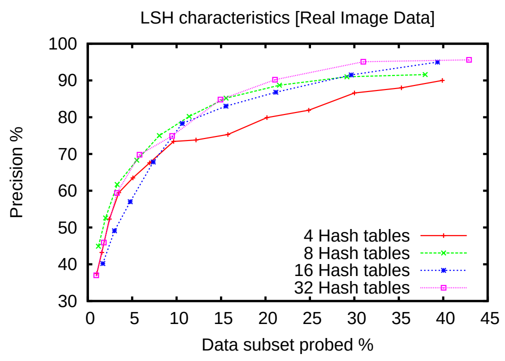

Figure 3: LSH characteristics on picture data.

tion of 1.702 seconds. Taking into account that each mapper has an approximately same data input size, defined by the HDFS' block size, approximately constant mapper execution time is well expected.

Measures of Interest

Because the execution time of the mappers is almost constant, the load of a query execution can be represented as number of mappers per query. We measured the number of mappers per query and precision for 50 KNN queries, with K=20, for both datasets, with 2GB, 4GB, and 8GB of indexed data (∼4000, ∼8000, and ∼16000 feature vectors for the real image dataset and ∼8000, ∼16000, and ∼32000 feature vectors for the synthetic dataset, respectively). The number of mappers per query for synthetic dataset is shown in Figure 5. And as we can see the number of mappers is about 3 times smaller for LSH than for linear scan. Also we can see in Figure 6, which shows the number of mappers per query for the Flickr dataset, that the difference in the number of mappers between LSH and linear scan is even bigger. The number of mappers per query is 4 to 5 times smaller for LSH than for linear scan in this case. The precision, shown in Figure 7, for generated data is over the threshold of 70% for 2GB and 4GB of indexed data, but drops down to 63.8% for 8GB. For real image data, the precision is almost constant, varying slightly around 86%.

6. CONCLUSION

In this work we described RankReduce, an approach for processing large amounts of data for K-Nearest Neighbor (KNN) queries. Instead of dealing with standard issues in distributed systems such as scalability and fault tolerance, we implement our solution with MapReduce, which provides these salient properties out of the box. The key idea of the presented approach is to use Locality Sensitive Hashing (LSH) in the Map phase of MapReduce to assign similar objects to the same files in the underlying distributed file system. While this seemed to be straight forward at first glance, there was a nontrivial conflict of opposing criteria and constraints caused by LSH and MapReduce which we had to solve to achieve accurate results with an acceptable query response time. We have demonstrated the suitability of our approach using both a synthetic and a real world dataset.

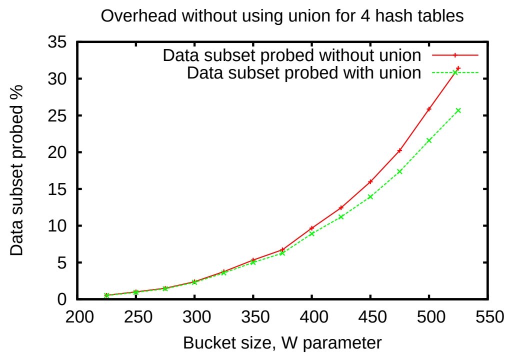

Figure 4: Overhead without using union.

Our presented approach on large scale data processing is, however, not limited to KNN search over images, but can be extended to a variety of other interesting applications, such as near duplicate detection, document classification, or document clustering.

As a first step in our future work plan to evaluate our approach on a real compute cluster, which we are currently building up, with large scale data in the order of several TB. We furthermore plan to extend our approach to video and music retrieval.

7. REFERENCES

[1] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, 2006.

[2] Michal Batko, David Novak, Fabrizio Falchi, and Pavel Zezula. Scalability comparison of peer-to-peer similarity search structures. *Future Generation Comp. Syst.*, 2008.

[3] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: self-tuning indexes for similarity search. In *WWW*, 2005.

[4] Jon Louis Bentley. K-d trees for semidynamic point sets. In *Symposium on Computational Geometry*, 1990.

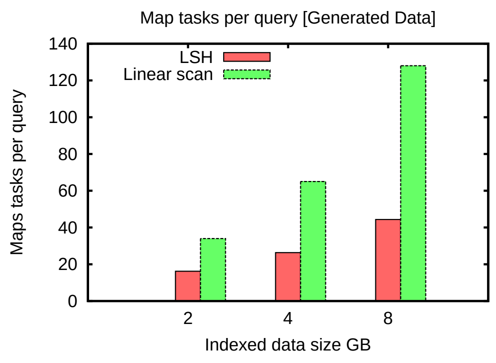

Figure 5: Map tasks per query on generated data.

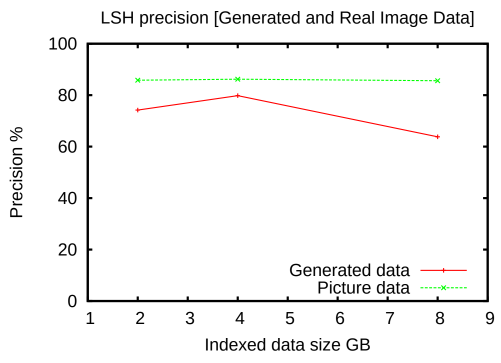

Figure 7: LSH precision on generated and picture data.

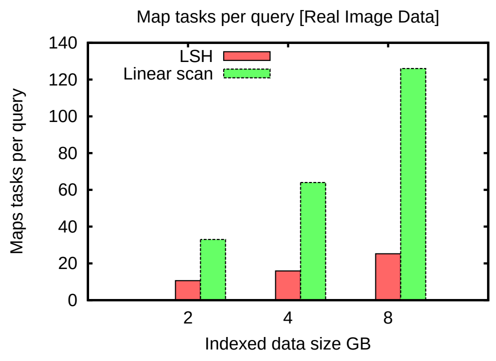

Figure 6: Map tasks per query on picture data.

[5] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The x-tree : An index structure for high-dimensional data. In *VLDB*, 1996.

[6] Kevin S. Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[7] Paolo Bolettieri, Andrea Esuli, Fabrizio Falchi, Claudio Lucchese, Raffaele Perego, Tommaso Piccioli, and Fausto Rabitti. CoPhIR: a test collection for content-based image retrieval. *CoRR*, 2009.

[8] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, 2004.

[9] Ritendra Datta, Dhiraj Joshi, Jia Li, and James Ze Wang. Image retrieval: Ideas, influences, and trends of the new age. *ACM Comput. Surv.*, 2008.

[10] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[11] Vlastislav Dohnal and Pavel Zezula. Similarity searching in structured and unstructured p2p networks. In *QSHINE*, 2009.

[12] Wei Dong, Zhe Wang, William Josephson, Moses Charikar, and Kai Li. Modeling lsh for performance tuning. In *CIKM*, 2008.

[13] Christos Doulkeridis, Kjetil Nørvåg, and Michalis Vazirgiannis. Peer-to-peer similarity search over widely distributed document collections. In *LSDS-IR*, 2008.

[14] Fabrizio Falchi, Claudio Gennaro, and Pavel Zezula. A content-addressable network for similarity search in metric spaces. In *DBISP2P*, 2005.

[15] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 2003.

[16] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[17] Parisa Haghani, Sebastian Michel, and Karl Aberer. Distributed similarity search in high dimensions using locality sensitive hashing. In *EDBT*, 2009.

[18] Shadi Ibrahim, Hai Jin, Lu Lu, Li Qi, Song Wu, and Xuanhua Shi. Evaluating mapreduce on virtual machines: The hadoop case. In *CloudCom*, 2009.

[19] Jimmy J. Lin. Brute force and indexed approaches to pairwise document similarity comparisons with mapreduce. In *SIGIR*, 2009.

[20] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, 2007.

[21] Richard M. C. McCreadie, Craig Macdonald, and Iadh Ounis. On single-pass indexing with mapreduce. In *SIGIR*, 2009.

[22] Vernica Rares, Carey Michael J., and Li Chen. Efficient parallel set-similarity joins using mapreduce. 2010.

[23] Ozgur D. Sahin, Fatih Emekçi, Divyakant Agrawal, and Amr El Abbadi. Content-based similarity search over peer-to-peer systems. In *DBISP2P*, 2004.

[24] Bilyana Taneva, Mouna Kacimi, and Gerhard Weikum. Gathering and ranking photos of named entities with high precision, high recall, and diversity. In *WSDM*, 2010.