Cloud-based Audio Fingerprinting Service

Jiang Wenyu*, Zhu Yongwei, Bao Xiaoming, Yu Rongshan
Email: {wjiang,ywzhu,baoxm,ryu}@i2r.a-star.edu.sg, *Correspondence author
Institute for Infocomm Research, Singapore

***Abstract*—Audio Fingerprinting allows the identification of a query audio clip by matching the query audio fingerprints against a reference database. Traditionally, the matching process, which is CPU and memory intensive, is implemented either on a single computer (which is confined by CPU and memory limits for large databases), or on a computer cluster in a proprietary manner (which has limited flexibility in scaling the database). We have implemented audio fingerprinting prototype software that can run in a Cloud environment, specifically Hadoop/MapReduce. Because the MapReduce framework is designed for stream data processing instead of database query, we discuss how we address this challenge as well as other challenges such as appropriate data input format and partitioning. A performance evaluation of the software on a real dataset of ˜8500 songs and real Hadoop clusters is presented to illustrate its efficacy, where a batch query of 1000 60sec clips can be completed in ˜50sec in addition to ˜30sec of database loading time with a 12-node cluster configuration.**

I. Introduction

Audio Fingerprinting is a technology that determines the identity of a query audio clip (where its metadata ID information may be missing or incorrect), by extracting content-based features from the clip, converting them to audio fingerprints, and compare them against audio fingerprints generated in the same manner from a database of reference content. It is useful in areas such as song identifiation, (near-)duplicate detection and content protection. In order to be scalable for a large reference database, this comparison must be implemented as an efficient search operation. Despite many progresses in improving the search efficiency [1], [2], [3], audio fingerprinting remains a task that is both CPU and memory intensive for a large database. Traditionally, it is implemented either on a single computer and thus confined by CPU and memory limits, or on a computer cluster in a proprietary manner which limits the flexibility in scaling the database.

To address the problems above, we have developed audio fingerprinting prototype software that can run in a Cloud environment, specifically Hadoop/MapReduce [4]. The choice of Cloud allows simple hardware scalability, while the MapReduce framework enables easy system scalability. Also, the Hadoop Distributed File System (HDFS, Chap. 3 of [4]) provides data location transparency, reliability and ease of data migration.

Although Cloud seems to be a straight-forward choice for scaling nearly any system, the MapReduce framework, despite its popularity, poses some unique challenges in the case of audio fingerprinting. MapReduce is designed for stream data processing, where output data are generated as input data come in [5]. In contrast, audio fingerprinting is essentially a database application, where output data (search results) cannot be generated until the database is completely loaded. In addition, there are other questions such as best approach to scaling such a system and data format compatibility, that need to be answered.

Our contributions in this paper are: First, we propose a solution to the conflicting paradigms between database and stream processing; Second, the pros/cons of several approaches to system scaling for audio fingerprinting are analyzed, particularly in view of MapReduce framework; Third, we have developed an audio fingerprinting prototype system and its performance is evaluated on real Hadoop clusters. To the best of our knowledge, this paper is the first study on Cloud-based audio fingerprinting.

The remainder of this paper are organized as follows. First, we go through a high level overview of MapReduce and its open-source implementation, Hadoop. Second, we describe the example audio fingerprinting algorithm that we aim to port to the Cloud, namely Google Waveprint [3]. Third, we describe our detailed solutions. Fourth, we present a performance evaluation of our prototype software on real Hadoop clusters.

II. Background

A. Hadoop/MapReduce Overview

In MapReduce, input data are presented as a stream of records, where a record is a key-value pair of type $< K1, V1 >$. A Mapper converts input data into intermediate data, which is also a stream of records (and also a key-value pair), although both its data types (of type $< K2, V2 >$) and the number of records may be different from input data. A large input data set may be partitioned *explicitly* into multiple files, or implicitly (i.e., *automatically* by MapReduce). Each partition is processed by exactly one Mapper (and one Mapper may process multiple partitions), and the number of Mappers control the level of parallelism. Intermediate data from all Mappers are merged into a single stream of data and then dispatched to one or more Reducers based on the key field in the intermediate data (i.e., $K2$). Each Reducer generates its share of output records (also some key-value pair of type $< K3, V3 >$), which are then collected as (final) output data.

Unless manually disabled, MapReduce assumes that data can be partitioned in any manner and the execution result should still be correct. This apparently strong assumption arises from the stream processing paradigm employed by MapReduce, i.e., each record (by default) can be independently processed. Later we will describe the example audio fingerprinting algorithm, Google Waveprint [3], and put into

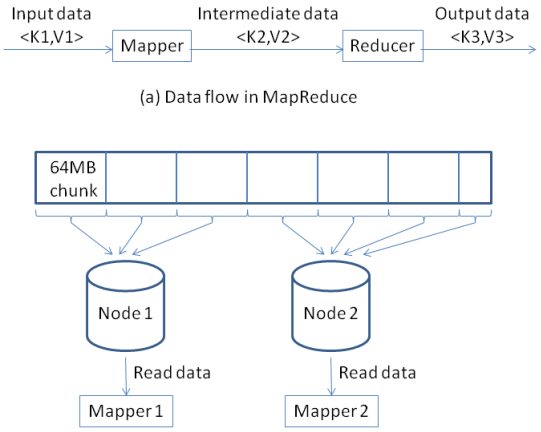

Fig. 1. Glance of Hadoop/MapReduce

perspective the specific challenges in porting this algorithm to the Cloud.

Hadoop is an open-source implementation of the MapReduce framework. It has its own file system, the Hadoop Distributed File System/HDFS (though some other FSes are also compatible with Hadoop runtime), which supports among other things data location transparency and fault tolerance to disk failure. A large file in HDFS is usually spread out in chunks (typically 64MB) and stored on hard disks of multiple nodes, so the Hadoop scheduler can execute a Mapper on a node that contains a local copy of the data whenever possible in order to reduce network traffic. Given that the chunk size is fixed, the larger a data file is, the higher parallelism Hadoop may support while minimizing network traffic.

Figure 1 gives a glance at Hadoop and MapReduce. Note for simplicity this example does not show data fault tolerance because each chunk has only 1 copy in HDFS. Also in practice consecutive chunks are more likely to be distributed on nodes in a round-robin fashion instead of on one node for multiple consecutive chunks.

B. Example Fingerprinting Algorithm: Waveprint

The specific audio fingerprinting algorithm we choose to port and evaluate on the Cloud is the Google Waveprint [3]. In brief, it extracts frequency information and presents them as a 2D artificial image of frequency histograms over time. Each 2D image is generated at a particular time interval (called a stride) and wavelets are then used to filter these images. The top-t (t=5%) wavelet coefficients in magnitude are retained and the positions and the +/- signs of these retained coefficients are lossy-compressed using Min-Hash [6], into p bytes (typically p=100) of signature information per stride. A signature, which is associated with a particular timestamp, is then divided into L hash keys for L tables. Typically L = 20 and the key size is 100/L = 5 bytes or 40-bit. The query audio clip goes through the same feature extraction and signature generation steps, except its stride is much shorter (typically 46ms) than the reference stride (typically 0.9s). The long reference stride is to reduce the number of indexed hash keys and hence reduce memory usage of the hash tables, albeit at an expense of more searches required with a short query stride to account for potential time offset between reference and query content.

With a hash-based search scheme, look-up is relatively fast but the query key has to exactly match the reference key in order to find the corresponding reference content. Even a single bit error would cause the right reference content to be missed, thus to improve recall multiple (e.g., L=20) tables are consulted hoping that at least one of L query keys will match exactly, though at an expense of higher memory usage. In addition, Waveprint tests for the number of matched tables on a matched reference signature for the L query keys (associated with the same query timestamp), and only if this number meets a minimum threshold v (chosen as 2 for typical configuration) would the match be considered further. This significantly reduces false positives (i.e., unrelated content that accidentally have the same key as query content), but has an important implication on data partitioning, to be detailed next.

Figure 2 summarizes the Waveprint algorithm. For convenience the default configuration parameters used in our evaluation are given in parentheses. This default is the Waveprint-1 configuration in [3].

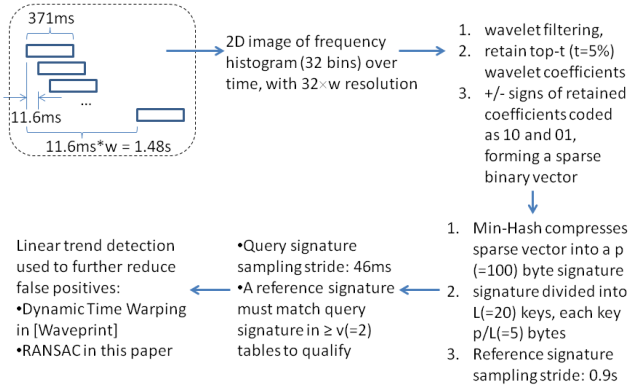

Fig. 2. Summary of Waveprint algorithm [3]

III. Challenges and Solutions in Porting to Cloud

In view of the Hadoop/MapReduce framework and requirements of Waveprint, the challenges and issues in porting Waveprint to the Cloud along with our proposed solutions are described next. For a preview, they are summarized in Table I.

A. Paradigm Mismatch

In audio fingerprinting, the entire database (e.g., the hash indexing tables if hashing is used for search) needs to be loaded before a search operation can begin. Therefore, the stream processing paradigm is not applicable. To tackle this problem within the MapReduce framework, we opt to treat the hash indices as a stream of data, where $K1$ is the hash key and $V1$ is the hash bucket. As the indices are read from hard disk, the Mapper loads the indices into memory instead

Challenges/Issues	Proposed Solutions (in Hadoop)
Paradigm mismatch - Database vs. Stream processing	(1) Make database look like stream: $K1$:= 64-bit hash key and $V1$:= the hash bucket of signature pointers as BytesWritable class; (2) Embed query in cleanup() when extending Mapper class
MapReduce processes terminate after task completion: Overhead in database loading	Execute query in large batch (e.g., 1000 60s query clips)
Efficiency of Data Partitioning: Horizontal vs. Vertical	Horizontal partitioning saves Mapper to Reducer network traffic, more flexible scaling
Proper Data Format: to ensure compatibility and prevent implicit partitioning	Prepare input data in Hadoop's SequenceFile format (binary), extend SequenceFileInputFormat class and overload isSplitable() to always return "false"

TABLE I
SUMMARY OF CHALLENGES/ISSUES AND PROPOSED SOLUTIONS

of generating any intermediate data. Once all the indices have been read from disk, the query can then commence, and to detect that in Hadoop, we extend the Mapper class and embed the query code inside its overloaded member function cleanup(), which is invoked by Hadoop scheduler only when all input data (of a partition) have been read. The query code generates the intermediate data, which is then collected by Reducer which generates the output data. The cleanup() function is specific to the Hadoop implementation of MapReduce, although it is conceivable that other MapReduce implementations may also have equivalent functions.

B. No Keep-Alive for MapReduce Processes after Task Completion

Note that even with the above approach, after Reducer is completed the entire indices will be unloaded from memory because all MapReduce processes will terminate after the job is complete. Since index building can take a significant amount of time for a large database, we choose to perform queries in large batches so that the amortized cost of index building per query is reasonable. In our experiments, such a batch consists of 1000 query audio clips of 60s each. At a query stride of 46ms, this corresponds to a total of around 1.3M query signature searches per batch.

C. Horizontal vs. Vertical Partitioning

For a single node case, the above explains the data flow well, whereas for multiple nodes, data partitioning and system scaling is more complicated. In general, a large database may be partitioned horizontally or vertically. In the case of audio fingerprinting, when using hash indexing, multiple hash tables are usually needed to achieve high recall as well as high precision. Therefore, vertical partitioning can be defined as choosing which hash tables belong to a partition, i.e., L hash tables are like L columns in a relational database. Conversely, horizontal partitioning is typically defined as which pieces of reference content should be in a partition. The difference of these two choices lie in their implications of performance, as detailed below.

For vertical partitioning, some hash tables are loaded in its entirety (i.e., across all reference content) on the same computer node. In Waveprint, if the L query keys at a particular query timestamp generates a hit for reference content signature C in only 1 table on that node, it could be that another hit for C may be generated for those L query keys on another node. Therefore, such a hit cannot be discarded and instead it must be reported as intermediate data, and the Reducers will determine whether at least v=2 tables are hit on C with these L query keys. This results in significantly more network traffic during the Reducing step. In contrast, with horizontal partitioning, all L tables are loaded on the same node, and if C is hit on only 1 table, then there is no chance that C could be hit in any table on any other node, because other nodes would not store indices for C (barring the existence of duplicates in reference database but that does no harm either) due to horizontal partitioning.

Note that if the Waveprint threshold v is set to 1 instead, then vertical partitioning would not suffer from unnecessary network traffic. However, with v=1, the false positives will be much higher, and the eventual network traffic (which cannot be avoided even with horizontal partitioning) will be about as high as that of vertical partitioning at v=2.

Therefore with horizontal partitioning, the network traffic to Reducer is minimized. Also, with vertical partitioning, as the database becomes very large, with a fixed L (e.g., 20), there can be at most L parallel nodes, whereas horizontal partitioning is much more flexible.

Also note that when performing the batch query, all query audio fingerprint files are read from the underlying HDFS explicitly, rather than being partitioned and each partition dispatched to certain node only. This is because each query thread (which has its own database partition) needs to read all query audio fingerprint files, so any network traffic due to reading query audio fingerprints cannot be eliminated whether in horizontal or vertical partitioning.

D. Data File Format and Preparation

The enforcement of partitioning may be explicit and/or implicit in Hadoop/MapReduce. If one very large file is used as input data, Hadoop may choose to split it virtually and launch one Mapper per split. This may cause a problem in audio fingerprinting, because to avoid any confusion the split point should not occur within a song. Because the choice of split point is not directly controllable by the Hadoop application user, we opt to create a special type of input file that is *Not* splittable, and we manually partition the input data into N such files, where N is the maximum desired number of Mappers. This is achieved by extending (i.e., deriving) a known Hadoop

input file format class (specifically SequenceFileInputFormat), and overloading its member function isSplitable() to always return "false". This subclass is then specified as the input file class by calling Job::setInputFormatClass() in the Hadoop main() function.

Typically, the input file is in text format and each line forms a record (where $K1$ is the line number and $V1$ is the line text). However, for compactness and efficiency, a binary format is desirable. For implementing Waveprint, the input file is designed to be an on-disk representation of hash table(s). Each record of a hash table is a pair of a 64-bit integer (hash key) and a variable-length byte sequence (hash bucket represented as a list of signature pointers). For simplicity, each signature pointer is a 32-bit integer with a fixed number of top bits being the reference content ID (e.g., song number) and the remaining bottom bits being the relative signature offset within that content. The file is prepared using Hadoop's SequenceFile format. Because the file should be non-splitable, all L tables should be bundled together in one input file during data preparation, so that each Mapper will read a complete partition consisting of all L tables.

IV. Implementation Details and Performance Evaluation

A. Data Creation

We used a database of 8464 songs with genres spanning rock, pop, classical, instrumental and multiple languages, and limited extraction to first 5 minutes of each song. First, Matlab code was written and used to extract frequency histograms and wavlet coefficients, followed by Min-Hash [6] to generate the 100-byte signatures at reference stride (0.9s) for each song, per the steps in Figure 2. Second, C++ code was written to read a given partition of songs, create L hash tables in-memory and then dump them as one file with a simple binary format: for each table, number of unique hash keys followed by pairs of key and bucket of signature pointers. Third, standalone [1] Java code was written to read the simple hash dump file and transform them into Hadoop's SequentFile format. Because Hadoop supports Java natively but not C++, it was more difficult to directly let the C++ code dump in the Hadoop SequenceFile format. Also, C++ was used in the second stage of data creation because it is more memory and speed efficient and easily customizable than Java. In particular, the hash buckets are constantly updated and realloc'ed during table building, which would have been much less efficient if handled by Java.

Note that because with horizontal partitioning all L tables reside on the same file, there needs to be a way to distinguish which table a pair of hash key and bucket belongs to. For code simplicity a dummy hash key is created with a MSB (most significant bit) of 1 and its remaining LSBs (least significant bits) representing table number i. This does not introduce any ambiguity because the key size is 40-bit, while the key stored on disk is 64-bit. For schema consistency, an empty dummy bucket is paired with the dummy key. Whenever such a dummy key is read, it indicates current table number has changed. Fourth, MapReduce Java code was written to read the hash indices in SequenceFile format and perform batch queries upon completing read each partition. Because horizontal partitioning allows maximum autonomy, a Mapper can start the query in its cleanup() function as soon as all data in its respective partition has been read, instead of waiting for other Mappers.

B. Linear Trend Detection

If the query audio clip has heavy distortion, its search results at each query timestamp will likely be noisy. To reduce both such noise and false positives, firstly for each search, only those content (more precisely, signature pointers) matched on at least 2 tables (i.e., v=2) are considered further as candidates; and secondly, these candidates are used for linear trend detection, and the query audio clip is considered to have matched a reference song only if a strong enough linear trend is detected among these candidates. Both criteria are shown in Figure 2.

RANSAC [7] is a simple way to detect linear trends in a large set of sample points even when the number of inliers is relatively small (e.g., 5-10%). It works by randomly selecting two points, measure their slope, and test for the number and quality of inliers along that slope. After a specified number of iterations, the slope with the best fit that meets some minimum degree of fit is chosen as final result. We use a slightly modified version of RANSAC, where a slope range of [0.9,1.1] is enforced to efficiently filter out many out-of-range slopes early on and thus save computation on inlier verification. Various thresholds are used to make sure the confidence of RANSAC detection is high. Note that RANSAC is performed in the Mapper, not the Reducer, because the needed data is already available at the Mapper, and doing it at the Mapper saves network bandwidth during Reducing stage. The intermediate data that the Mappers generate are RANSAC slope detection results, which are copied by the Reducer as-is for output data. Therefore for this implementation, most of the intelligent logic reside in the Mapper, and the Reducer is basically a dummy class. RANSAC only uses a small portion of total execution time.

C. Performance Results

We evaluated our prototype software on an internal 3-node (1 name node + 2 tracker nodes [2]) Hadoop cluster with each node having dual quad-core CPU (Xeon E5310 1.6GHz), and on an external 15-node (1 name node + 14 tracker nodes) Hadoop VM cluster where each VM having dual core (2.66GHz). In both clusters, each node has 4GB RAM and runs CentOS 5.5 32-bit. The 8464 song database was horizontally partitioned to 12 parts that are roughly equal in size. The Hadoop version is 0.21.0 on both clusters, it was

[1] In the sense that it is run by Hadoop, but not using the typical Hadoop main() function structure, hence not invoking the Hadoop scheduler.

[2] The name node is responsible for assigning jobs to tracker nodes and managing HDFS and usually does not run the actual jobs

Cluster (# name nodes + # tracker nodes)	Node Particulars	Common Configurations	DB loading vs. Query time (per node)	Query time average (per node)	Total elapsed time
3-node (1+2) internal	dual quad-core Xeon E5310 1.6GHz	Hadoop 0.21.0; CentOS 5.5 32-bit; 4GB RAM & $\leq$ 2 parallel Mappers per node; database with 12 horizontal partitions; JVM heap size=1.5GB	20-30s; 25-55s	37.2s	222s
15-node (1+14) external VM	dual core 2.66GHz			33.2s	78s

(a) Hadoop related configurations and performance results

Name	This paper	c.f. [3]
Database	8464 songs (limited to 5min), 2.25M keys per table	10000 songs (3.5min avg), 2.3M keys per table
Query setting	1000 60s clips @ MP3 32kb/s, 46ms query stride	same
Accuracy	992/(1000-3) = 99.5% (3 have wrong ground truth), time offset at (-0.2s,1.6s) error range	99.15%, no reported error range

Reducer output	RANSAC results with format & sample output below
	`query clip#/matched song# (slope,intercept) ...`
	`00000,11 (1.024,164.5) w 220 / 408 inliers, ranked 0`
	`00001,25 (1.027,89.3) w 219 / 432 inliers, ranked 0`
	`...`	

(b) Database-related performance results

TABLE II
SUMMARY OF EXPERIMENTS

configured to allow up to 2 parallel Mappers per node, and since the database has 12 parts, this implies 4 and 12 parallel Mappers on internal and external clusters, respectively.

1000 60s query audio clips are randomly selected from the database in terms of both song name and relative time offset, and various distortions are applied to test the audio fingerprints' robustness. Because the main purpose of this paper is to evaluate Cloud applicability rather than audio fingerprint robustness, for brevity we only describe the results on the distortion introduced by MP3 32kb/s compression, which is the MP3-32K test case in [3].

For convenience, the experiment results are summarized in Table II. Our main findings are: First, cloud-based audio fingerprinting is feasible. The database loading time, i.e., time until Mapper::cleanup() is called, is 20-30sec for each partition. After that, the query takes about 25-55sec to run for each partition, with the average 37.2s and 33.2s for internal and external clusters, respectively. This makes the database loading overhead reasonable vs. the query time. Note the internal cluster is slower due to slower CPUs, and it runs at most 4 parallel Mappers, thus its elapsed time is about 12/4 = 3 times as long as the external cluster, as seen in Table II(a).

Second, our RANSAC implementation of linear trend detection works well compared to the performance reported by [3], which has its own proprietary method using Dynamic Time Warping. Out of 1000 query clips, 992 are successfully identified with both the right song name and *the right time offset* (with a -0.2 to 1.6sec error range). In comparison, [3] does not elaborate whether the right time offset criterion is enforced. Out of 8 undetected query clips, 3 happened to have the wrong ground truth (its audio fingerprint and suggested reference song name doesn't match), thus the identification accuracy is 992/(1000-3) = 99.5%, which is better than the 99.15% reported by [3] on the same MP3-32K distortion test. Besides the difference in song database, the only difference is the linear trend detector. However, the detector still needs further improvement on query clips 30s and shorter. Out of 5 remaining unidentified query clips, 4 of them are highly repetitive tunes and thus linear trend detection would intuitively be difficult. It is unknown whether the song database in [3] contains such highly repetitive tunes.

Third, the Java Virtual Machine (JVM) needs to be configured with a large enough heap size (1.5GB instead of the default 0.5GB) under the 12-partition setting. The specific configuration file is $HADOOP_HOME/conf/mapred-site.xml The Java implementation appears to be much less memory efficient than C++, because the database in [3] is 10000 songs at 3.5min average length and is similar in database size, estimates $<$ 0.5GB memory usage for the entire database, whereas our Hadoop Java code requires up to 1.5GB (likely around 1GB) memory for 1/12 of the entire database. We believe this is due to the inefficiencies in memory allocation in JVM.

One other difference between our implementation and [3] is the omission of full signature comparison. Because the reference signatures are not automatically loaded during database loading, it appears this would add significant overhead (either much higher memory usage if preloading all raw fingerprints, or slow disk I/O otherwise) if full comparisons are implemented. Therefore, we omit this step and instead rely on the $\geq$ v test and linear trend detection to filter out the false positives.

V. CONCLUSIONS AND FUTURE WORK

Cloud computing provides flexibility in scaling a system, but to realize and maximize that flexibility, the system in question needs to be analyzed for its requirements and particulars. In the case of audio fingerprinting, its database-centric

nature is at odds with the stream processing paradigm of MapReduce. Fortunately, this apparent conflict can still be resolved in the Hadoop implementation of MapReduce, by (legitly) "hacking" APIs that are not typically used. This enables database processing, but queries must be invoked in large batches to limit database loading overhead, because MapReduce processes holding the database will by definition terminate after task completion. By examining a specific audio fingerprinting algorithm like Waveprint, we gain more insights into the trade-offs between horizontal and vertical partitioning for system scaling, concluding that the former is much more efficient and flexible. Also, to enforce partitioning properly, data must be prepared in a suitable format. These are summarized in Table I.

By following the above approaches to porting audio fingerprinting to the Cloud, we have implemented prototype software and evaluated their performance on two Hadoop clusters. The results confirm feasibility of Cloud-based audio fingerprinting. With a batch size of 1000 60s query clips and a ˜8500-song database with 12 partitions, database loading time (20-30s) is comparable to query time (averaging 33.2s and 37.2s per partition for the two clusters respectively). Our MapReduce code was written in Java as it is the preferred language in Hadoop. It shows reasonable CPU-time efficiency, but its memory efficiency is a lot poorer, seemingly requiring 20-30x more memory than C/C++ would have.

Our implementation, being slightly different from [3] mainly in linear trend detection, demonstrates good accuracy, at 99.5% vs. the 99.15% in [3].

In the future, we plan to examine a wider spectrum of scalability results e.g. with respect to the configured number of parallel Mappers. We also hope to explore the use of C++ in Hadoop and quantify its speed and memory benefit over the current Java implementation.

ACKNOWLEDGMENT

The authors would like to thank Chan Ti Eu at Institute of Infocomm Research for helping to set-up the VM Hadoop Cluster for our experiments.

REFERENCES

[1] J. Haitsma and T. Kalker, "A highly robust audio fingerprinting system," in *International Conference on Music Information Retrieval (ISMIR)*, 2002.
[2] P. Cano, E. Batlle, T. Kalker, and J. Haitsma, "Audio fingerprinting: concepts and applications," *Studies in Computational Intelligence (SCI)*, vol. 2, pp. 233–245, 2005.
[3] S. Baluja and M. Covell, "Waveprint: Efficient wavelet-based audio fingerprinting," *Pattern Recognition*, vol. 41, pp. 3467–3480, 2008.
[4] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 2009.
[5] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, 2004.
[6] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. Ullman, and C. Yang, "Finding interesting associations without support pruning," *Knowledge Data Engineering*, vol. 13, no. 1, pp. 64–78, 2001.
[7] Wikipedia, "Random sample consensus (ransac)," http://en.wikipedia.org/wiki/Ransac.