

The Architecture and Algorithms of Audio Indexing

B vk.com/blog/arhitektura-i-algoritmy-indeksatsii-audiozapisey

Why is it necessary?

We have a lot of music, over 400 million recordings which altogether add up to 4 PB. This is as if you downloaded all music from VK onto 64 GB iPhones, then stacked them one on top of the other, the result of which being a tower higher than the Eiffel. Every day another 25 iPhones need to be added, or 150,000 new recordings totaling 1.5 TB.

Of course, not all of these files are unique. Each file has data about the performer and title (sometimes lyrics and genre), which a user fills in while uploading a song to the website. Since there is no pre-moderation, we receive different titles, remixes, concerts and studio recordings of the same songs, as well as completely misnamed recordings.

If we learn enough about how to find the same (or very similar) recordings, it can be used to one's benefit, such as:

- Not double one recording with different titles in the search results.
- Recommend listening to favorite songs in HQ.
- Add cover images and lyrics to all song variants.
- Enhance the recommendation mechanism.
- Improve work with the requirements of content owners.

Perhaps the first thing that comes to mind is ID3-tags. Each .mp3 file has a set of metadata and we can take into account this information as more a priority than what the user indicated in the website interface while uploading the recording. This is the simplest solution, although it's not the best one. Tags can be edited manually and do not need to necessarily correspond with the content.

So all the information associated with the file that we have depends on people and may be unreliable. This means we need to work with the file itself.

Thus we set ourselves to the task of defining recordings that are the same or similar in sound by analyzing only the contents of the file.

It seems somebody has already done this?

Searching for similar audio recordings is a very popular story. Using acoustic fingerprints is already a classic solution used by everyone from Shazam to biologists studying wolf howling.

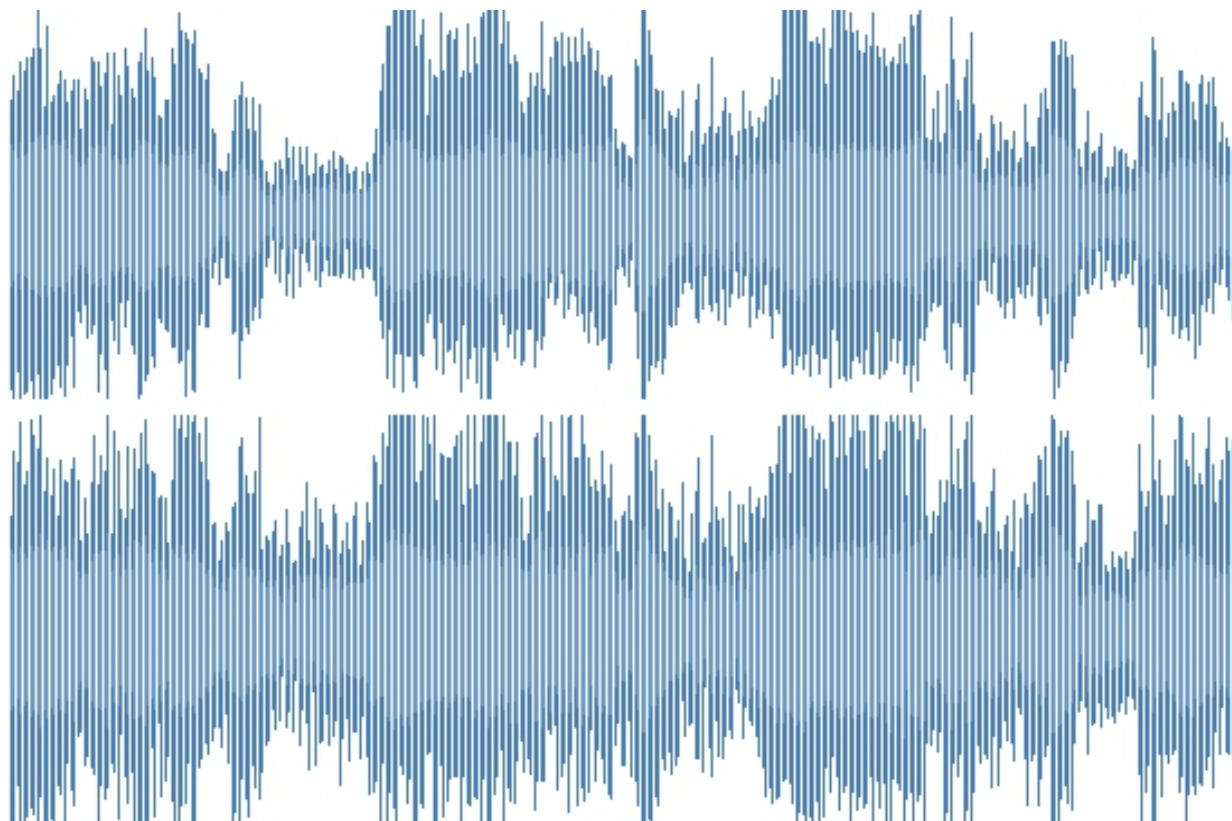
Acoustic fingerprints are a representation of an audio signal in the form of a set of values that describe its physical properties.

Simply speaking, a print contains some information about the sound. Moreover, this information is compact, much smaller than the original file. Songs that sound alike will have similar fingerprints and, vice versa, different sounding songs have contrasting fingerprints.

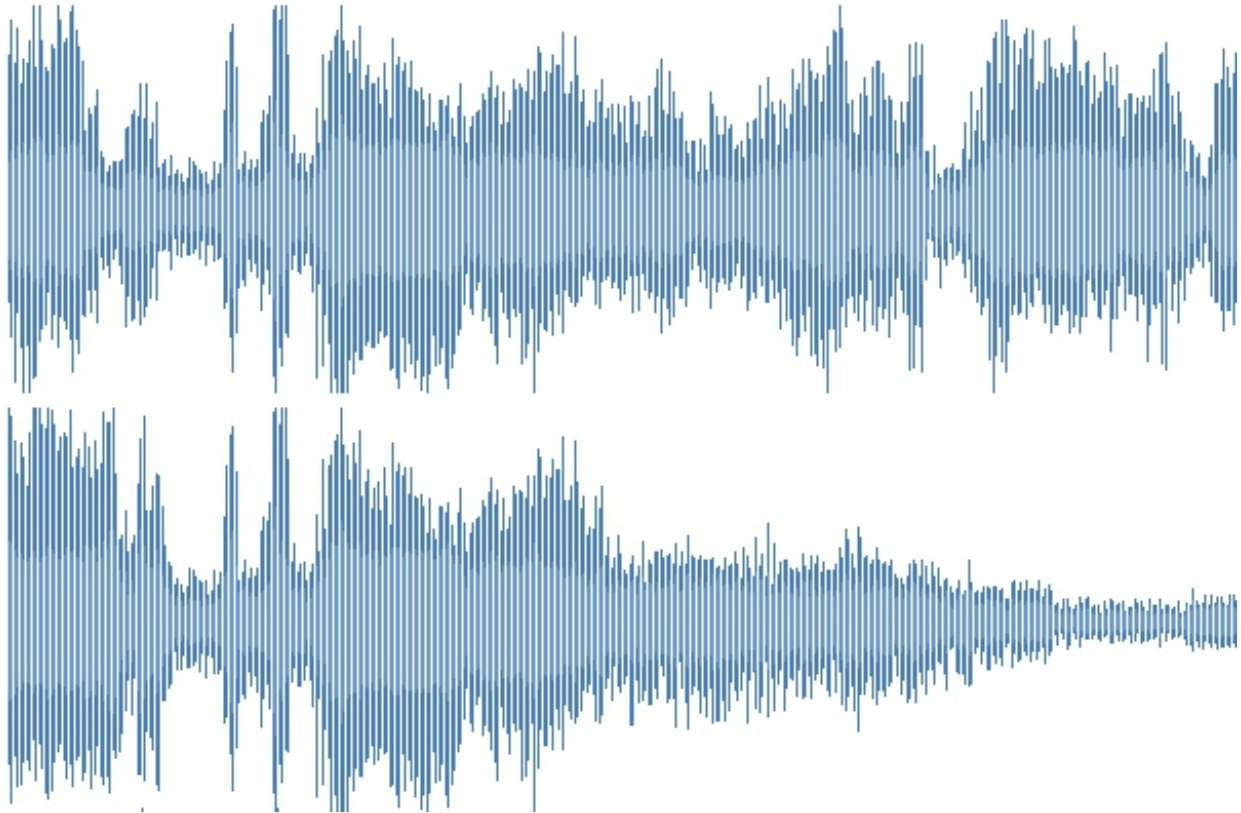
We began by attempting to use one of the existing solutions in C++ to generate acoustic fingerprints. After fastening it to our search and testing it on real files, we understood that the results were poor for a large part of the sample.

The same track is successfully “masked” by equalizers when extra background noise or jingles are added, or mixing with another track.

Live performance



Echo



Remix

In all these cases, a person easily understands that this is the same song. We have many files with similar distortions, so it is important to be able to get good results for them. It became clear that we needed to create our own code to generate fingerprints.

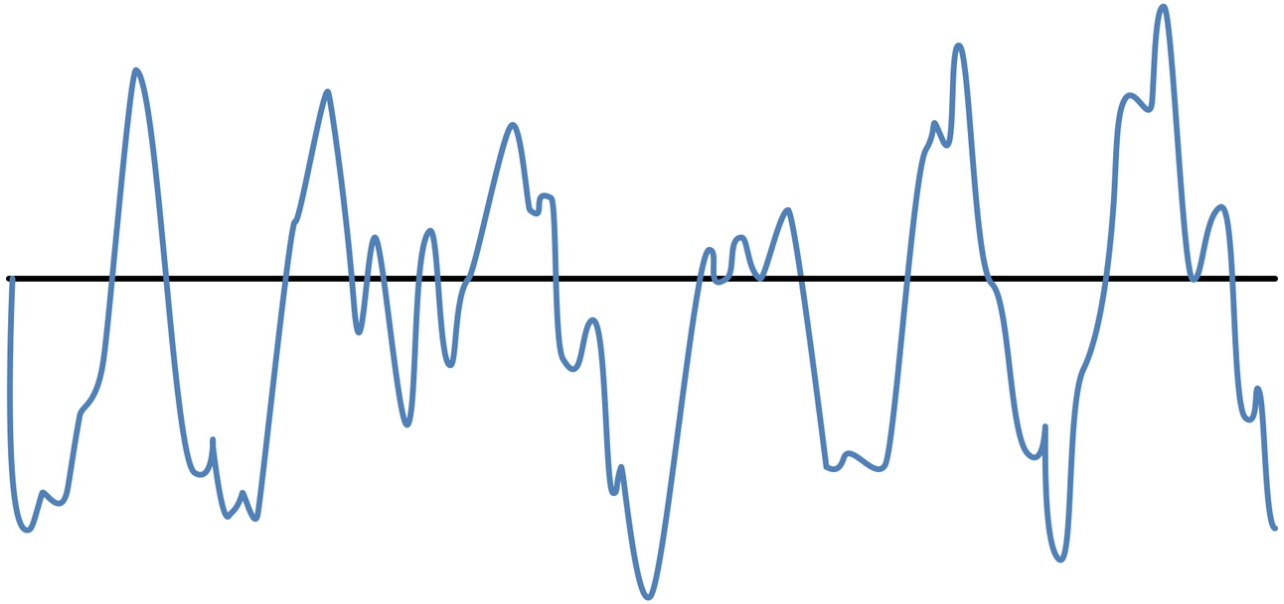
Fingerprint generation

Imagine we have an .mp3 audio file. How can we transform it into a compact print?

To start with, we have to decode the audio signal that was packed into the file. MP3 presents itself as a chain of frames (blocks), which contains encoded data about audio in a PCM format (pulse code modulation). This is uncompressed digital audio.

In order to acquire PCM from MP3, we used the library [libmad](#) in C and our [own wrapper](#) in Go. Later, we opted for the direct usage of [ffmpeg](#).

One way or another, in the end we have an audio signal in the form of array of values describing the dependence of amplitude over time. One can imagine it as a graph:



Audio signal

This is the sound that our ears hear. A person can perceive it as whole, but in fact a sound wave is a combination of numerous sound waves consisting of different frequency waves. This is akin to a musical chord, which consists of several notes.

We want to know which frequencies exist in our signal, especially which of them are the most characteristic for it. Let's try the canonical way of obtaining such data, which is using a fast Fourier transform (FFT) algorithm.

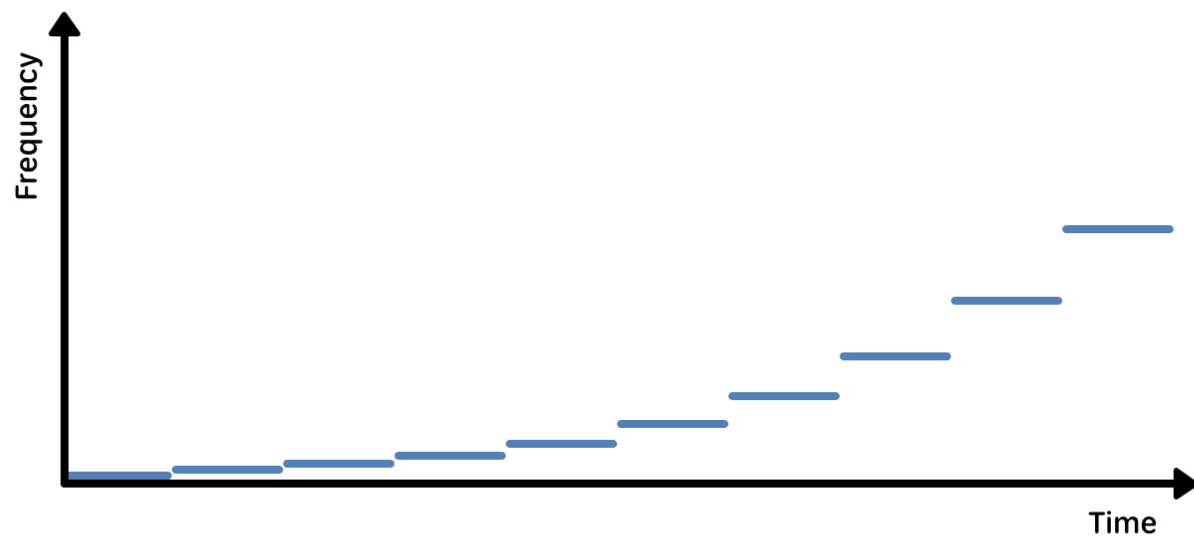
Describing this mathematical apparatus will take longer than this article. To learn more about using FFT in the field of digital signal processing, you can read [this publication](#).

In our realization, we use the [GO-DSP \(Digital Signal Processing\)](#) package, in particular <http://github.com/mjibson/go-dsp/fft> — actually FFT and <http://github.com/mjibson/go-dsp/window> for the Hann window function.

At the end, we receive a collection of complex numbers, which is called a spectrogram when it is transferred to a plane.

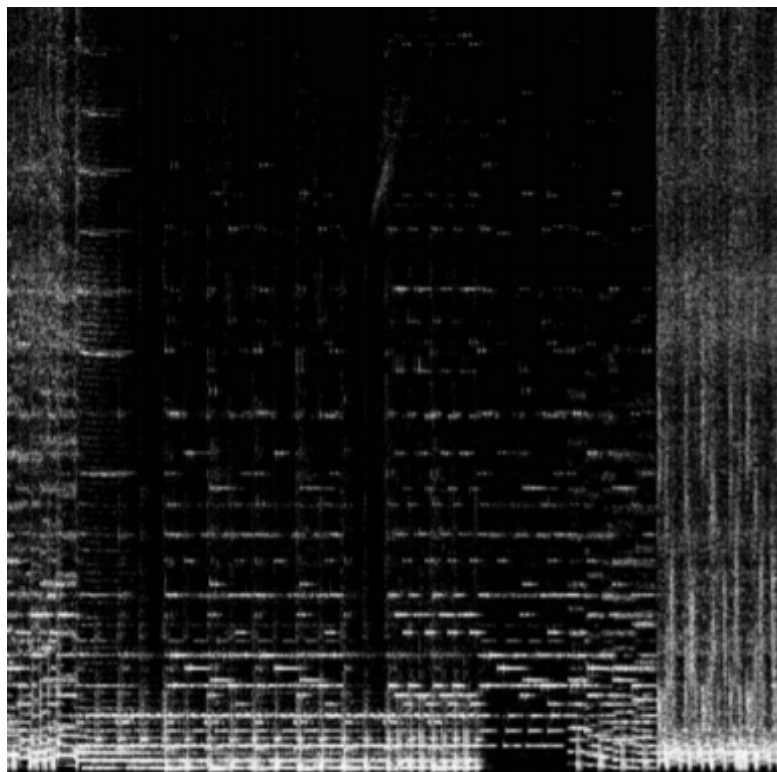
Spectrograms are visual representation of all three acoustic dimensions of a signal: time, frequency and amplitude. It defines the amplitude for a particular frequency at a certain moment in time.

For example:



Reference spectrogram

Time is measured along the X-axis, while the Y-axis represents frequency. The amplitude is portrayed by pixel color intensity. On the illustration, a spectrogram is provided showing the “reference” signal with equally increasing frequency. For an average song, a spectrogram looks like this:



Regular spectrogram

This is a fairly detailed “portrait” of the audio track, from which one can (with a certain approximation) restore the

original track. Considering resources, it is unprofitable to store a “portrait” in its full size. In our case, this would require at least 10 PB of memory.

We choose key points on the spectrogram (based on intensity of the spectrum) to save only the most characteristic values of this track. As a result, the volume of data decreases around 200 times its original size.

Key values on a spectrogram

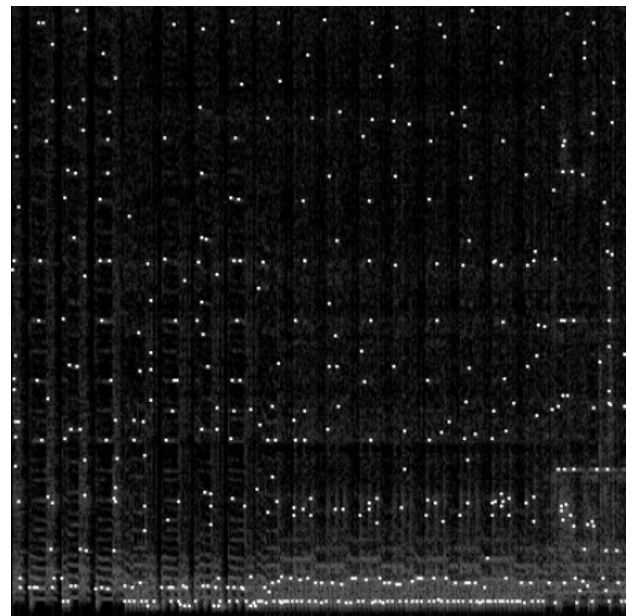
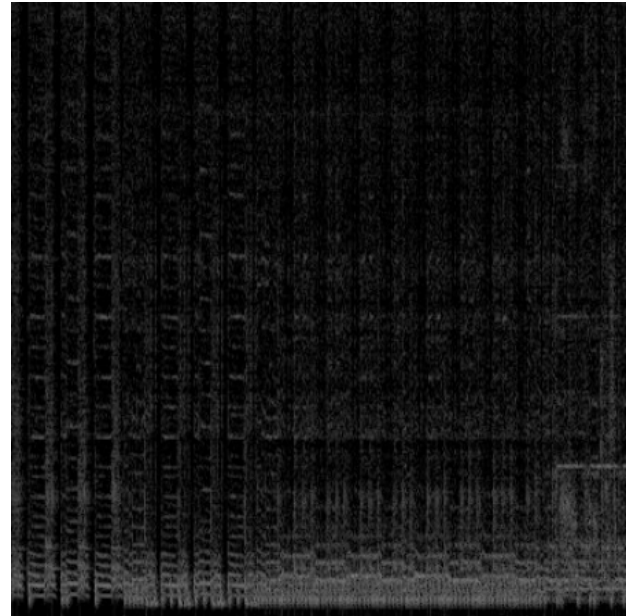
What is left is to collect this data in a convenient format. Two numbers uniquely determine each pick: frequency and time. By including all the peaks of a track into one array, we receive the desired acoustic fingerprint.

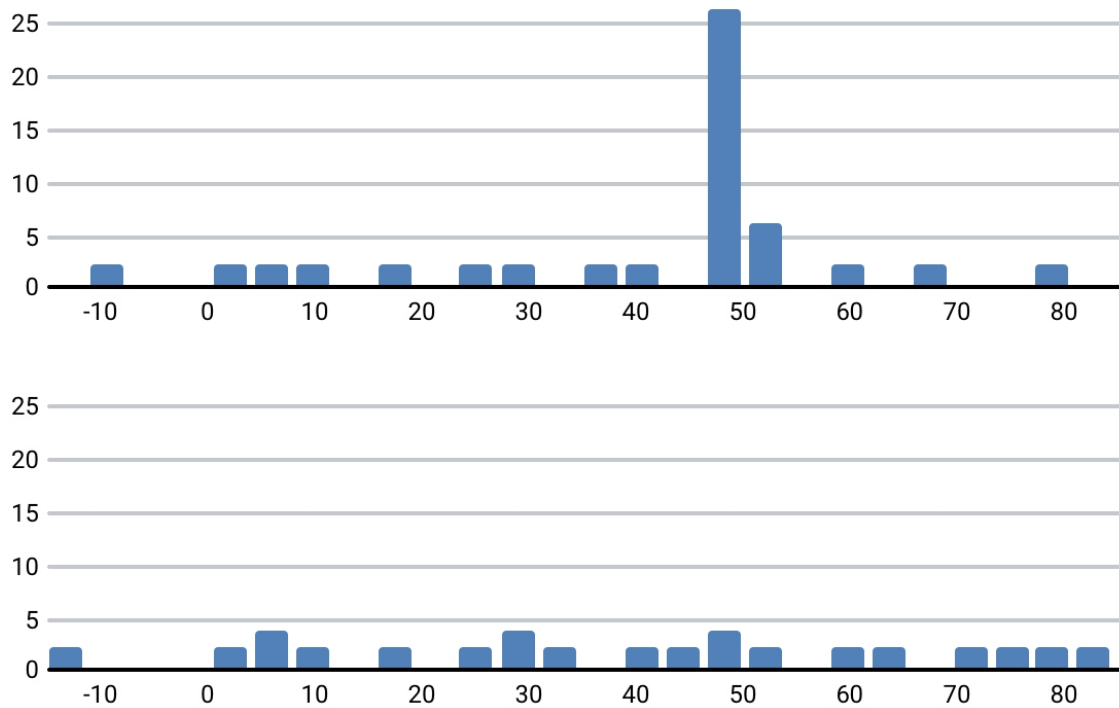
Comparison of fingerprints

Assuming we completed all previous steps for two hypothetical songs, and now we have their fingerprints. Returning to the original task of comparing these tracks with the help of fingerprints to determine whether or not they are similar.

Each fingerprint is an array of values, and we try to compare their elements, moving songs along the time scale relative to each other (a move is needed, for example, to take into account the silence at the beginning or the end of a song). In some cases, there will be more coincidences in the fingerprints while others will have fewer.

It looks something like this:





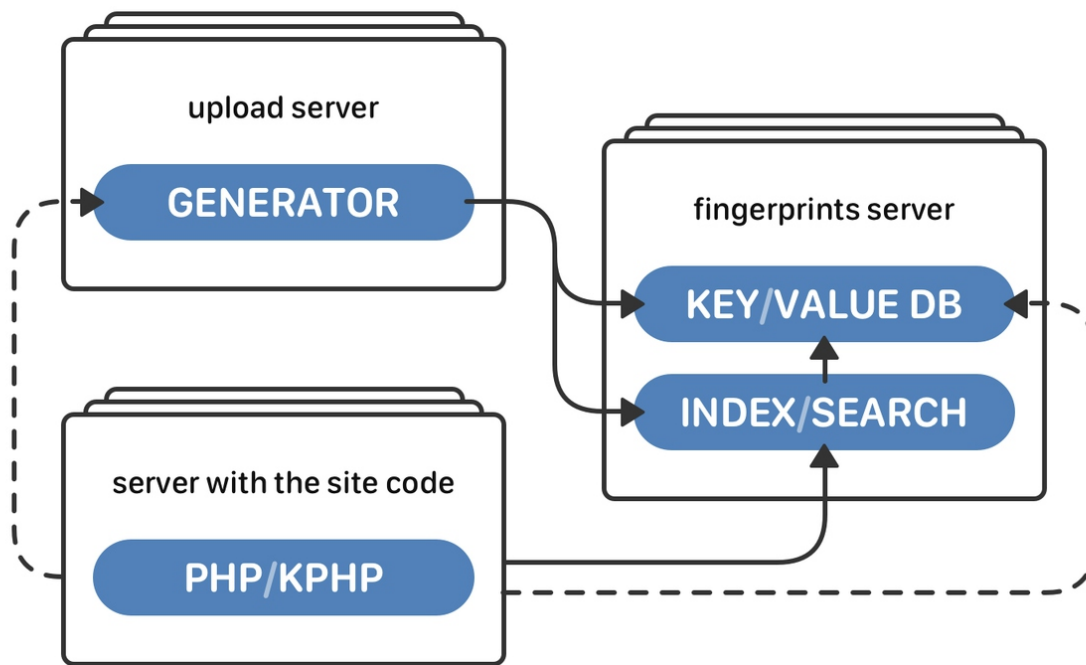
Tracks with a common fragment and different tracks

It seems correct. For tracks with a common fragment, the fragment was found and resemble the peak of a number of coincidences at a certain instant. The result of this comparison is the “similarity coefficient”, which depends on the number of coincidences after taking into account the bias.

The software implementation of the Go library for generating and comparing fingerprints is available [on GitHub](#). You can see the graphs and results for your own examples.

Now it is necessary to integrate everything into our infrastructure and see how it goes.

The Architecture




Fingerprints Generators and Search/Index Engines in VK Architecture

The engine for generating fingerprints works on every server that uploads audio (now there are currently about 1,000). It takes an .mp3-file, processes it (decodes, FFT, highlights spectrum peaks) and produces an acoustic fingerprint of this audio.

The load is parallelized at the file level where each track is processed in a separate goroutine. For an average song of 5-7 minutes in duration, processing takes 2-4 seconds. This processing time increases linearly based on the duration of the audio.

Acoustic fingerprints of all tracks, though with the loss of some clarity, will take about 20 TB of memory. The entire volume of this data needs to be stored somewhere and has to be applied to quickly in order to find something located inside. This task is solved by a separate index and search engine.

The engine stores the data regarding fingerprints in the form of reverse indexes:

Hashes	Tracks IDs						
0	A	C	E	M	W	X	
1	B	D	N	O	S	Q	
3	C	G	I	M	P	R	
7	A	D	E	K	O	W	
							
100500	F	K	P	R	V	Z	

Reverse index

To achieve speed and economize memory, we take advantage of the fingerprint's own structure. A fingerprint is an array, and we can consider its individual elements (hashes), which, if you remember, corresponds to the spectrum's peaks.

Instead of storing a "track" → "fingerprint" correspondence, we divide each fingerprint into hashes and save the "hash" → "list of tracks where it is located in fingerprints" match. The index is decimated, and 20 TB of fingerprints as an index take about 100 GB.

How it works in practice? A request with an audio recording arrives to the search engine and all similar tracks must be found. An audio fingerprint for this track is then downloaded from the repository, and lines containing this fingerprint's hashes are collected within the index. From the corresponding lines, frequently selected recordings are chosen for fingerprint scanning from the repository. These fingerprints are compared to the fingerprints of the original file. As a result, the most alike recordings with corresponding matched fragments and relative "similarity coefficient" are returned.

The index and search engine runs across 32 machines and is written in clear Go. It is here that goroutines, internal worker pools and parallel work with both the network and the global index are used to their maximum.

At this point, now that all the necessary logic is ready, fingerprints can be collected, indexed and worked with. But how long will it take?

We started indexing, then waited a couple of days and estimated the timescales. In the end, the entire process would take about a year. Such a long time is unacceptable, so something has to be changed.

Implementing sync.Pool anywhere possible shortens duration by two months, leaving 10 months total, which is still too long.

Optimizing the type of data - that is choosing songs according to their index was accomplished by merging the array. Using instead a container/heap saves another six months. But can it be better?

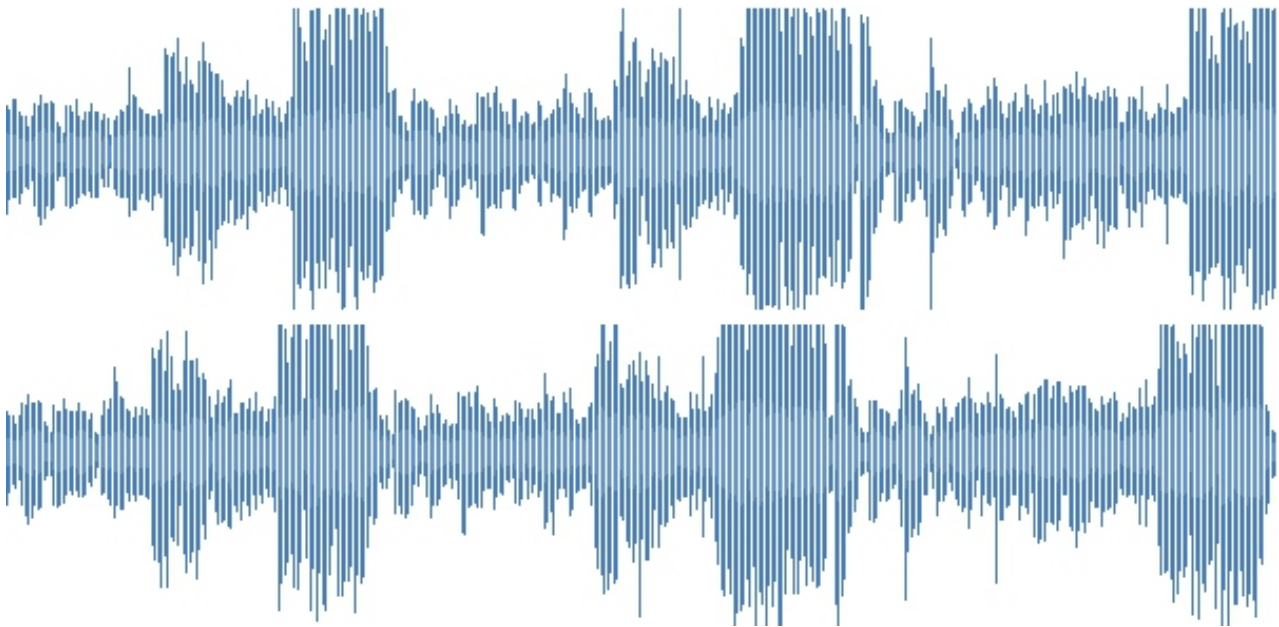
Adjusting container/heap to work with our data type instead of standard interfaces wins us an additional month. But it's still not enough.

We adjusted stdlib, by making its own implementation for container/heap, earns two more months, thereby bringing us to three. Four times less than the original estimate!

And finally, updating Go from 1.5 to 1.6.2 brought us to the final result of 2.5 months, the amount of time required to create the index.

What happened?

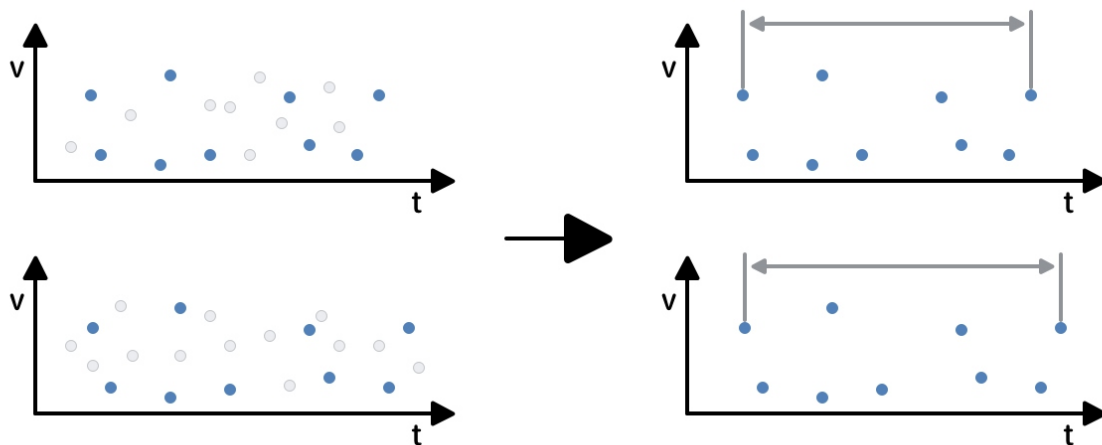
Production-testing revealed several cases that we did not take into initial consideration. For example, a copy of a track with a slightly adjusted playback speed:



Accelerated track

To the listener, this is nearly identical – a small acceleration isn't perceived as significantly different. Unfortunately, our algorithm for comparing fingerprints considered such an edit as quite different.

An additional amount of preprocessing was added to fix this. It is the search for the longest common subsequence in two fingerprints. If amplitude and frequency do not change, then only the corresponding time value changes in this case, and the common order of points following one after the other is preserved.

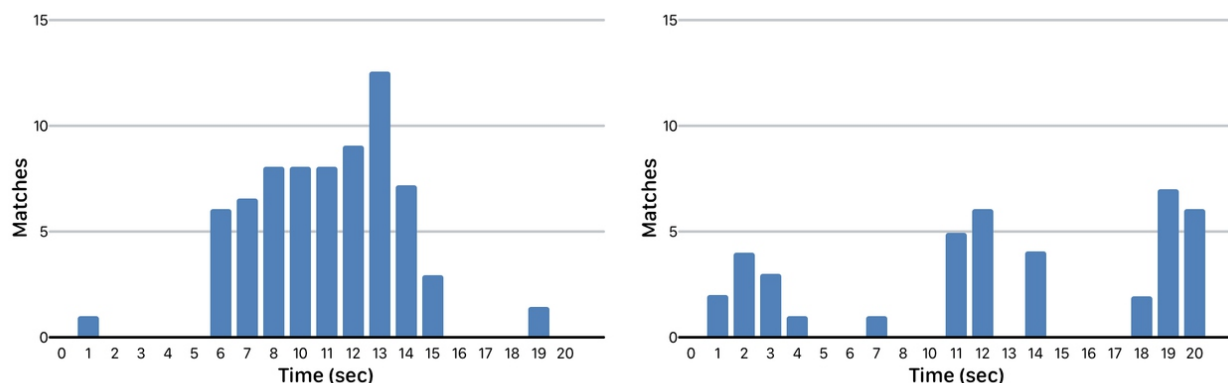


LCS

Finding LCS allows for the determination of the coefficient of “compression” or “stretching” affecting the signal’s time scale. Next, prints were compared as usual by applying the determined coefficient.

The use of the LCS search algorithm significantly improved the results by successfully processing many tracks that had not previously been searched by fingerprints.

Another interesting case is the coincidence of fragments, such as recording amateur vocals over the instrumental of a popular song.



Matching track fragments

We decomposed comparison results by time and assessed the number of matches for each second of a track. In the picture above is an example of an amateur recording over the instrumental track. Intervals with no coincidence – vocals, coincidence peaks – are silent (i.e. clearly similar to the original recording). In this situation, the number of fragments with coincidences was considered and the conditional “similarity coefficient” was calculated based on the number of coincidences.

After clustering similar tracks, individual clusters turned out to be much larger than the others. Why? There are interesting situations that are not very understandable regarding how they are correctly considered. For example, everyone knows ‘Happy Birthday to You’. There are dozens of variants of this song, which differ only in the name of

the recipient. Should they be considered different or not? The same goes for versions of the song sung in other languages.

These distortions and combinations of them became a serious problem during the launch stage.

We had to considerably refine the mechanism to take into account these problems and other uncommon situations. Now the system works as it should, including on such complex clusters, and we improve it each time we encounter new unique transformations.

Aside from such rare exceptions, our decision proved viable. Songs can be found when accelerated, vocals are removed, listed under fictitious names and jingles are inserted. The task was accomplished, and, undoubtedly, more than once it will be useful for future developmental work on the recommendation service, music searches and the audio section in general.

Have a question?

You may ask the author your questions [here](#).