FullContact

Photo Deduplication

Team:
Collton Deskin
John Kelly
Michael Lewis

Introduction

The Client

"If you've got an address book, you've got an address book problem." Contact information is often collected in fragments, leaving the user to assemble and manage it by hand. FullContact aims to change that by automating the storage, assembly, and retrieval of fragmented contact information. A user of FullContact's web service (currently in beta) is able to import and merge contacts from various sources (physical business cards, email lists, social media, etc.), and then later retrieve their contacts from anywhere in the world.

The Product

As part of the contact conglomeration process, duplicate data is generated. Removing duplicate (and even just similar) text is a classic and relatively easy problem in computer science. However, identifying visually similar images deterministically is not such a trivial task. Presenting multiple duplicate image choices for representing a contact to the end user makes for a poor user experience.

Thus, a process is needed which will identify visually similar images for removal, and this process must be highly scalable to large data sets, on the order of billions of inputs. In addition, Precision must remain perfect, or nearly so, while achieving maximum Recall possible (see Appendix A for definitions of Precision and Recall).

By automating this process well, FullContact will be able to significantly improve its user experience by selecting an image to represent a contact. The system will appear to be merging contacts (i.e. removing extraneous information), rather than simply collecting them together.

Requirements

High Level Description

As part of the dataset abstractly referred to as a "contact" in FullContact's system, a set of avatars (images representing a particular digital account) is maintained. Because these images are pulled from multiple related sources, it is very possible for a contact to accumulate one or more duplicate images.

The purpose of the project was to design a software solution which can identify such duplicate images accurately enough to provide a useful function to the end user. This required the evaluation and implementation of several image comparison algorithms to decide which provided the best balance of performance and correctness.

Functional Requirements

The Photo Deduplicator needed to be able to recognize photos that are “the same” with a high degree of Precision and Recall.

Specific Functionality included:

- Fetching contact pictures
- Comparing contact pictures
- Must have high degree (>50%) of Precision and Recall
- Must be highly scalable
- Must be able to run on Amazon’s Elastic MapReduce
- Must be able to process images from Amazon’s S3

Non-Functional Requirements

- Must not hinder UX
- Must be built using either Gradle or Maven
- Must have 70% unit testing coverage
- Team must decide whether to do Deduplication upon retrieval of photos, upon capture of the photos, or as a batch process in the background.

Risks

- Technical
  - The algorithm selected for identifying duplicate images may scale poorly on large datasets
  - The parameters of the algorithm may not have settings which will produce high Precision/Recall without considerable processing
- Skill
  - Pre-existing team experience with HTTP and RESTful APIs was low (one member with experience)
  - No member on the team had experience with Maven or Gradle
  - No member on the team had experience with scalable server technologies (Hadoop, Storm, etc.)
- Other
  - May have run into problems with licensing (specifically with “viral” GPL code)

Design

Core Library

In order to easily deduplicate images in a specific FullContact address book contact, the team produced a library which is available and deployed on the FullContact maven repository. The library exposes the following high-level functionality:

PhotoFetcher

PhotoFetcher is the base interface for acquiring image data given a set of URLs.
Implemented By:

- **AsyncPhotoFetcher**: Uses asynchronous HTTP requests to fetch images quickly.
- **DiskCachingPhotoFetcher:** Uses a local disk cache to stabilize fetch returns.

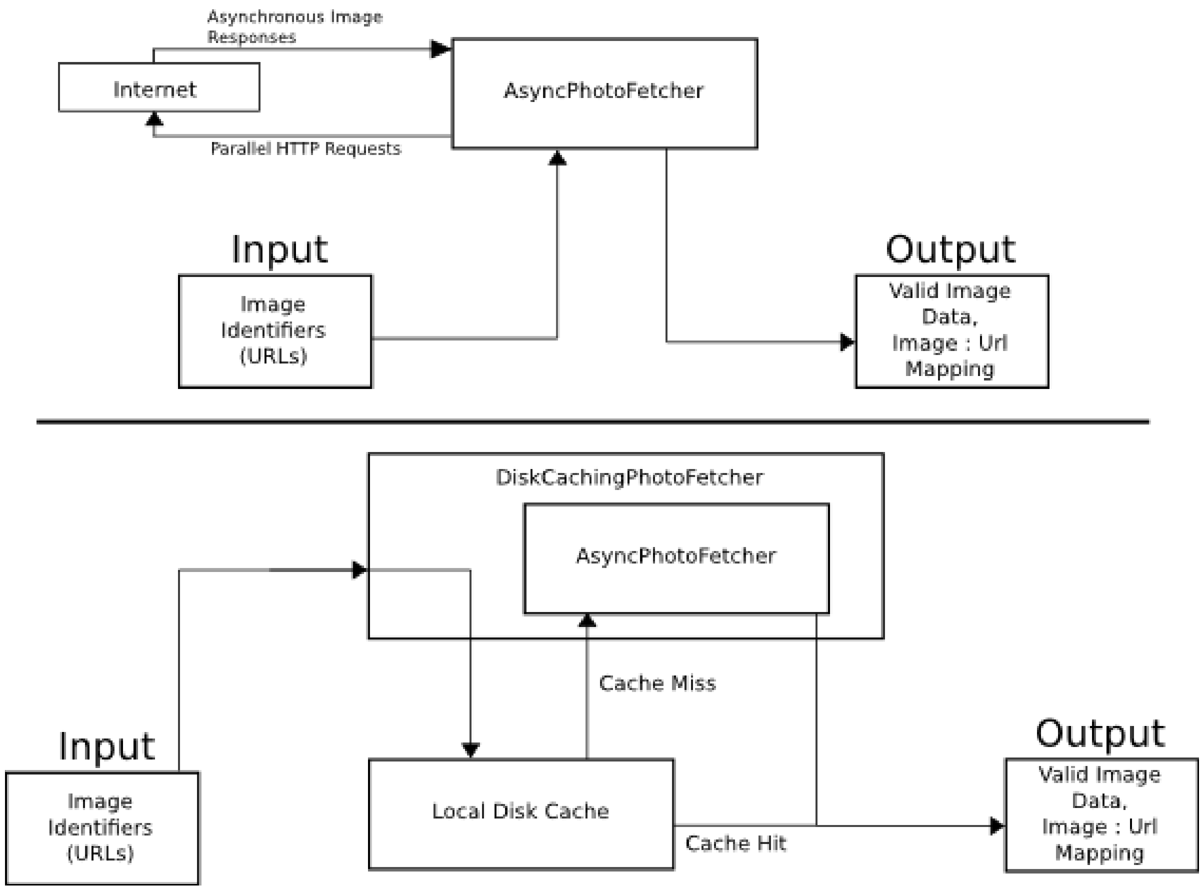


Figure 1 : Photo Fetching Lifecycle

<u>ImageComparisonGenerator</u>

The ImageComparisonGenerator receives a set of images and returns a data structure that contains each image as well as each of the images that are similar to it and their comparative similarity (hamming distance). In order to perform this action, it implements a strategy pattern using the ImageHash interface. See Figure 2, below.

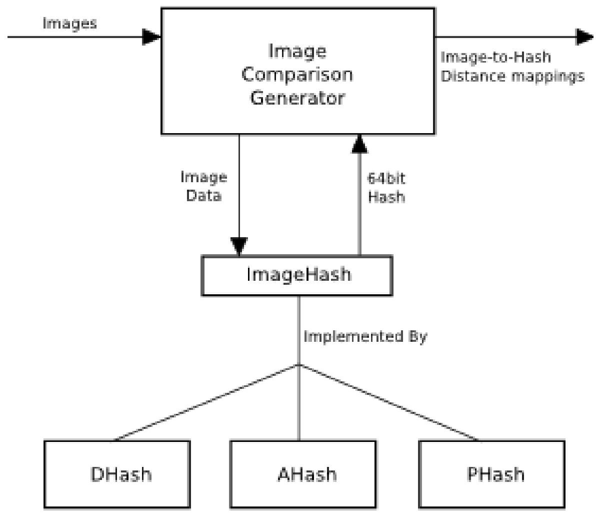


Figure 2: Image Comparison Generator Data Flow

<u>ImageScaler</u>

The ImageScaler class takes a given image, reduces it to a chosen size, and grayscales it. The ImageScaler is also able to extract and return luminosity levels of a given image, another common operation in image hashing algorithms.

<u>ImageHash</u>

The ImageHash interface receives a BufferedImage Java object and returns a 64-bit (i.e., Java type long) hash based on the chosen hashing algorithm. Example algorithms include AHash (average luminosity based hash), DHash (luminosity gradient based hash), and PHash (discrete cosine transform based hash). Additional algorithms can be extended from the interface if necessary for performance or correctness.

Implemented By:

- **PHash**: Hashes based on the result of a discrete cosine transform
- **AHash**: Hashes based on the average luminosity of the image
- **DHash**: Hashes based on the changes in luminosity of the image

(See Appendix B for more details on algorithm specifics)

FastDCTMaker

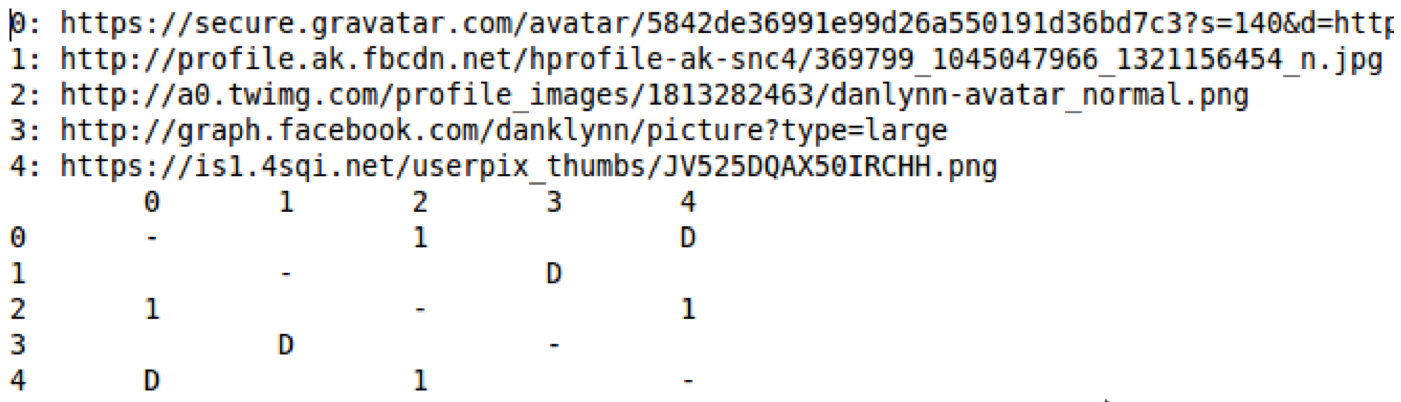
The naive solution to performing a DCT has $O(n^4)$ complexity on the size of the input image. The FastDCTMaker implements an optimized solution to the DCT calculation which is only $O(n^3)$ complexity, thereby presenting a significant speedup in the PHash calculation.

Profiling Subpackage

This package is a small set of classes segregated off from the core functionality of the library. These classes are intended to provide simple, out-of-the-box demonstration of library functionality and usage.

UrlDeduplicator

The UrlDeduplicator parses the output from ImageComparisonGenerator into a user-readable, tab-delimited, relational matrix to show duplicates within a threshold. The UrlDeduplicator is included in the Profiling subpackage as a demonstration of the library's capabilities of remote image fetching and hash comparison.



```
0: https://secure.gravatar.com/avatar/5842de36991e99d26a550191d36bd7c3?s=140&d=http
1: http://profile.ak.fbcdn.net/hprofile-ak-snc4/369799_1045047966_1321156454_n.jpg
2: http://a0.twimg.com/profile_images/1813282463/danlynn-avatar_normal.png
3: http://graph.facebook.com/danklynn/picture?type=large
4: https://is1.4sqi.net/userpix_thumbs/JV525DQAX50IRCHH.png
        0       1       2       3       4
0       -               1               D
1               -               D
2       1               -               1
3               D               -
4       D               1               -
```

Figure 3: Example UrlDeduplicator Matrix

StatisticsGenerator

The StatisticsGenerator takes an argument to specify which hashing algorithm to use, as well as a file of human generated comparisons. The StatisticsGenerator reads this file and compares each pair of URL's to see if they are duplicates. It then uses the given indicator to determine if it's results were correct. While comparing images, the StatisticsGenerator keeps track of how many true positives, false positives, true negatives, and false negatives it finds and uses these to calculate Precision and Recall for the given hashing algorithm at each comparison threshold. See Appendix B for example outputs used in the final selection of a hashing algorithm for this project.

Hadoop Functionality

An important aspect of the project was scalability. Running a Hadoop MapReduce job was chosen for this purpose. As part of a Hadoop job, a small number of classes needed to be developed with specific functionality. For more information about Hadoop, see the Technical Design section.

ImageInputFormat

The ImageInputFormat is a custom Hadoop InputFormat that is used to define how the Hadoop job will accept input files. The ImageInputFormat specifies the ImageRecordReader as the default Hadoop RecordReader for the Hadoop job. The ImageInputFormat also specifies that input files are not to be split and are to be processed as whole files. (Images are stored as discrete files on the HDFS)

ImageRecordReader

The ImageRecordReader does the bulk of the processing in the MapReduce job. The ImageRecordReader receives a split from the ImageInputFormat, in this case a full image file, and passes to the mapper a <key, value> pair consisting of the S3 image URI and a hash value.

HashMapper

HashMapper is a part of a Hadoop job designed to run on Amazon's Elastic MapReduce service. The mapper receives a URI to an image in a given Amazon S3 database and a hash for that image. The mapper then stores this information in text files in the specified output file for use by the HashReducer. Each node in a Hadoop cluster running Amazon's m1.large machines has 3 map threads so for a cluster with 40 machines, there will be 120 mappers running simultaneously and storing information in the output folder.

HashReducer

There is generally 1 reduce thread per node in a Hadoop cluster so there are up to 40 reducers running simultaneously in a 40 node cluster. Each of these reducers is reading input from the map outputs and, in our case, simply storing that again into one file. This reduces the number of files to process. If all map outputs were run through a single reducer, it would produce a single output file with the results of all the processed images.

Technical Design

Hadoop

The use of Amazon's Elastic MapReduce and other Web Service utilities allowed the team to scale the small scale library into something that could be used on extremely large datasets with relative ease. The final Hadoop job involved a 42 node Hadoop cluster with each node in the cluster running 4 threads simultaneously. Three threads on each node were dedicated to mapping while one thread on each node was dedicated to reducing. This allowed the team to process a high volume of images with great speed. In the end, it took approximately 50ms per image (with startup and shutdown overhead averaged across all images) to fully process and produce the desired output for 100,000 images. This could be sped up even more by increasing the number of nodes in the cluster, by increasing the number of threads per node, or by designating more of the processing to the various nodes. This makes the system very scalable, which is very important for FullContact's eventual goal of using the library to process their 2 billion image database. Figure 4 shows how data is handled in each node of a Hadoop cluster.

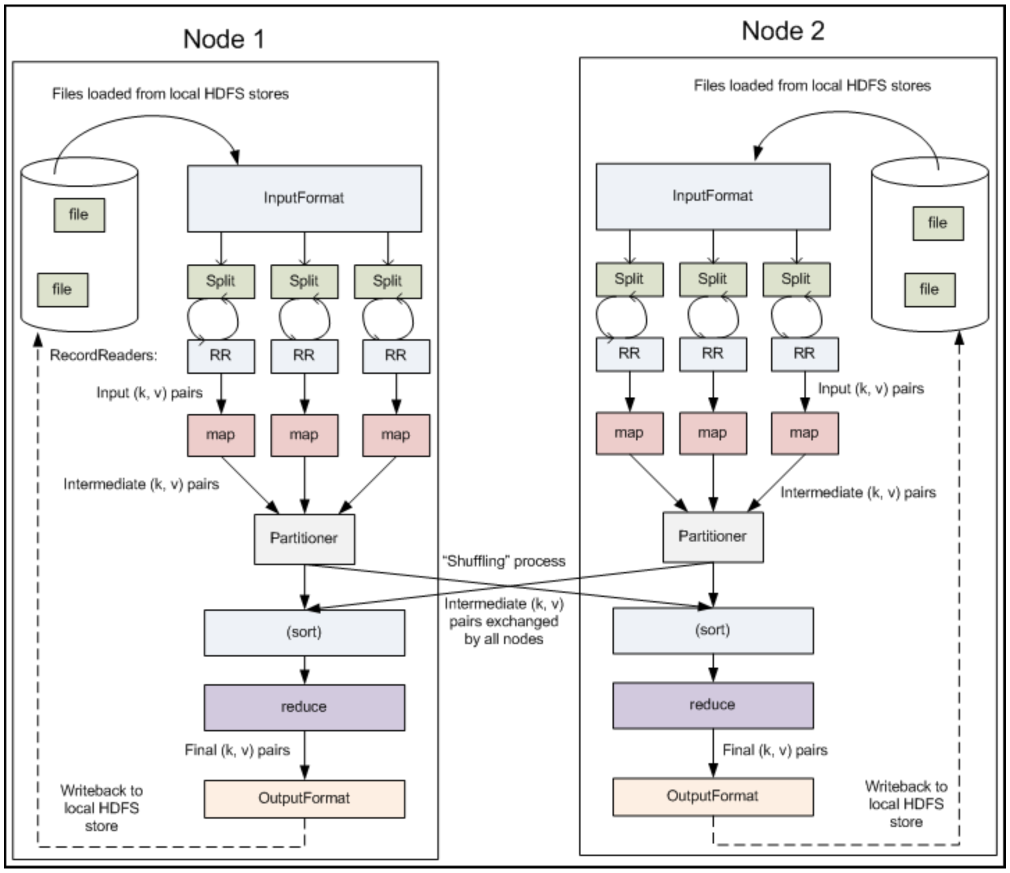


Figure 4: Hadoop Data Flow

An Efficient DCT

The 2D DCT calculation as part of PHash posed an interesting algorithmic problem. The naive implementation performs poorly enough that PHash attains an undesirable performance-to-speed ratio. As part of the team's research into the problem, an implementation that was only O(n^3) complexity (compared to the naive O(n^4)) was discovered. However, the efficient implementation came without comments or documentation, and was written in inexpressive code. A key nonfunctional requirement of the project was high testability to maintain code health, so a "black box" algorithm was unacceptable.

After much analysis and whiteboarding, it was determined that the efficient implementation was correct. A small amount of additional algebraic analysis demonstrated why it was correct. A 2D DCT relies on the collection of several different frequencies of visual data at various strengths, as determined by the input image, accumulated in two orthogonal dimensions. In the naive implementation, these frequencies are treated as unrelated to the destination pixel in the resultant DCT and instead a position-dependent factor is applied to each such pixel; however, a small amount of clever algebra reveals that these factors can be subsumed into the frequency terms to form output-dependent frequencies. By cleverly transposing indices in the calculation body, rows and columns of pixels can be collected into the DCT in a single iteration while maintaining the location dependency of their coefficients, thus significantly reducing the complexity of the algorithm. Although the full derivation of this result is far too lengthy for this report, it can be accomplished with only a moderate amount of linear algebra and time.

Nonstandard Color Spaces

As part of the large scale Hadoop job setup, it was discovered that Java's native ImageIO library has trouble reading certain images. Specifically, in this case, it was crashing the job when it attempted (and failed) to read jpg images that were encoded in unusual color spaces. Most jpg images will use the sRGB or Adobe RGB color profiles and convert internally to the more efficient YCrCb color space; however, theoretically any color space with a matching color profile that allows for conversion to and from YCrCb can be used.

When the specification for an image's color profile is not included or is included in a non-standard method, the ImageIO library is unable to parse the image data and throws an exception. The team investigated several solutions, most of which were unsuccessful at reading the selected test images for this edge case, and eventually settled on a particular algorithm which would handle several relatively common CMYK color spaces at the cost of significant code bloat and low readability.

Serendipitously, the chosen algorithm proved so difficult to implement that the job was run with the unusual color space code disabled, which led the team to discover the actual proportion of images that were unreadable by the ImageIO library. Less than 0.2% of all images (margin of error 0.015%) were unreadable by the simple solution. Although the client was interested in lowering this proportion, it was agreed that the diminishing returns were so low as to be unreasonable to pursue at this time.

Design Decisions

Identifying duplicate images is a fuzzy logic task. It is easy for humans but non-trivial for computers. The team needed an algorithm that would generate a signature for an image which would vary relative to the amount of modifications made to the image. This takes the form of a hashing algorithm that is resistant to small changes in image input data, unlike traditional hashes such as MD5, where hashes are distributed essentially randomly across input data. The three primary hashing algorithms that the team investigated were Average Hash (AHash), Difference Hash (DHash), and Perceptual Hash (PHash). Of the three options, PHash exhibited noticeably better Precision and Recall at a cost of increased running time. This increase in runtime was caused by PHash's larger seed image. PHash stores and processes 16 times as many pixels as AHash and approximately 14 times more pixels than DHash in order to compute it's 64 bit hash. The increase in computation time caused by PHash was negligible, however, compared to the computation time taken in scaling and grayscaling the images, which was common to all three of the hashing implementations. As such, PHash was chosen for the final hashing implementation.

With respect to the choice of language, the client requested that the team use either a Ruby or Java derivative not only to keep the utility and design environment consistent with the FullContact setting, but also so that the knowledge and experience of the FullContact development team had with these languages would be available and relevant. The team decided upon the use of standard Java because it offered efficiency as well as easy integration into various build systems (including testing libraries). On top of this each team member already possessed some measure of experience with Java. Ruby was a highly considered alternative due to its simpler handling of files and input streams, however the team decided against it, because there were a smaller number of FullContact employees who were dedicated Ruby programmers (compared to dedicated Java programmers) suggesting possible integration complications and difficulty finding help when needed.

The team chose to use the Gradle build system, which is based on Groovy (a Java derivative), to compile and build the jar files for the library. There is a scripting element to the build system, as well as extensive plug-ins which allow for compilation by convention and simplification of build specifications. Gradle was also more strongly supported within FullContact’s development team than alternatives such as Ant or Maven. This meant that help was easily accessible. Gradle’s scripting style was also very similar to Java, making it more natural to learn on the fly. To support the automated build system of Gradle, the team chose to use FullContact's Jenkins server to automate the code testing process; on top of this, a Cobertura plugin was applied to profile test coverage. The team did not consider other automated testing options because FullContact already had a Jenkins server which could be easily integrated into. Putting all this together, the team was able to setup automated building and testing which allowed more time for coding rather than testing, building, and verifying.

As part of the original project specification, the client requested that the image deduplication implementation have very high scalability. This led to implementing a MapReduce style algorithm to work with Amazon's S3 and Elastic MapReduce systems to handle high volumes of images from an S3 storage server. Using Amazon Web Services allowed the team to avoid manual setup of a Hadoop cluster, which would be well beyond the scope of the project.

Due to performance concerns, the team also chose to use a Code Profiler to detect hot-spots and repeated access to sections of code. For this, YourKit was chosen at the suggestion of the client. From this the team was able to deduce which function calls in the library were taking the longest in order to identify potential optimizations to maximize performance. Unfortunately, the heaviest computational tasks were in Java's native image scaling functions, and were beyond the team’s control; team-written functions accounted for less than 10% of all processing time.

Results

At the core of the project the team’s objective was to create a Java library that would allow a user to generate and compare hash values for given images. This small scale library performed above expectations, surpassing the 50% targets in Recall and Precision with ease. Figure 5 demonstrates the performance of PHash, which was selected as the best hashing algorithm of the three that were implemented. The most important of the two metrics, for FullContact’s use cases, is Precision because it is important not to erase pictures from a contact that are not, in-fact, duplicates. PHash was able to sustain Precision of over 99% up to approximately 83% Recall and Precision over 90% up to approximately 90% Recall [Figure 5].

The only concern with the PHashing algorithm was an increase in runtime due to the more intense computation involved in performing a discrete cosine transform on a larger image than the other two implementations. PHash was still able to produce approximately 500 hashes per second locally, however, so it was decided that runtime could be sacrificed for the better Precision and Recall metrics provided by PHash.

The core library is also under a test suite with approximately 89% total conditional coverage. Conditional coverage is important in test coverage because it implies coverage of all possible control flows. In addition to the core library, there were also several testing and profiling utilities which were used to test functionality and demonstrate capabilities of the library. These utilities included a file system based image deduplicator, a web based image deduplicator, and a statistics generator. Once large scale testing was started, the team discovered an issue with Java's ImageIO library that does not allow it to load images encoded in certain color spaces, or images produced by certain programs or cameras. Producing a robust image loader that would handle all color spaces and encodings was beyond the scope of the project, but after running an Elastic MapReduce job ignoring images that were unloadable, the team found that only about 100 of the 113,800 images in our test set were missed. The test data set was a selection of the first 100,000 images in FullContact's database. This meant that the images in the test set were uncorrelated so the team is confident that more than 99.9% of the images in FullContact's database of close to 2 billion images will be processable.

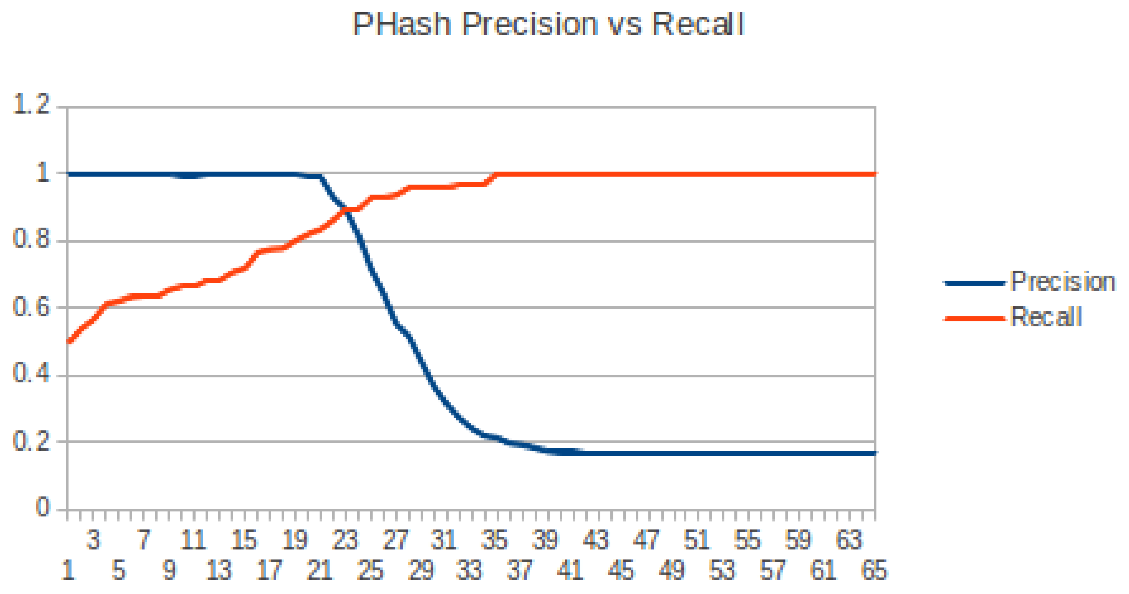


Figure 5: PHash Precision vs Recall

Appendices

Appendix A: Precision and Recall

Precision - Precision is a measurement of how much of the returned data is relevant and is given by the equation (True Positives)/(True Positives + False Positives).

Recall - Recall is a measurement of how much of the relevant information was retrieved and is given by the equation (True Positives)/(True Positives + False Negatives).

Appendix B: Image Hashing Algorithm Details

PHash

The PHash algorithm scales and grayscales a given image down to a 32x32 square. It then performs a discrete cosine transform on the stored luminosity data in the image which compresses low-frequency (highly human-visible) visual data into the upper left quadrant of the image. After averaging this information it then generates a 64-bit hash by comparing each pixel of the upper left 8x8 quadrant with the average value of the discrete cosine transform. See Figure A.1.

AHash

The AHash algorithm converts each image that it receives to a grayscale, 8x8 pixel thumbnail. Once this is done, it calculates the average luminosity of the thumbnail. AHash then proceeds through each pixel and generates a hash based on whether each pixel is above or below the average. See Figure A.1.

DHash

The DHash algorithm converts each image that it receives to a grayscale, 9x8 pixel thumbnail. It then generates an 8x8 matrix of directed gradients between pairs of pixels, oriented from left to right, which are processed into a 64-bit hash. See Figure A.1.

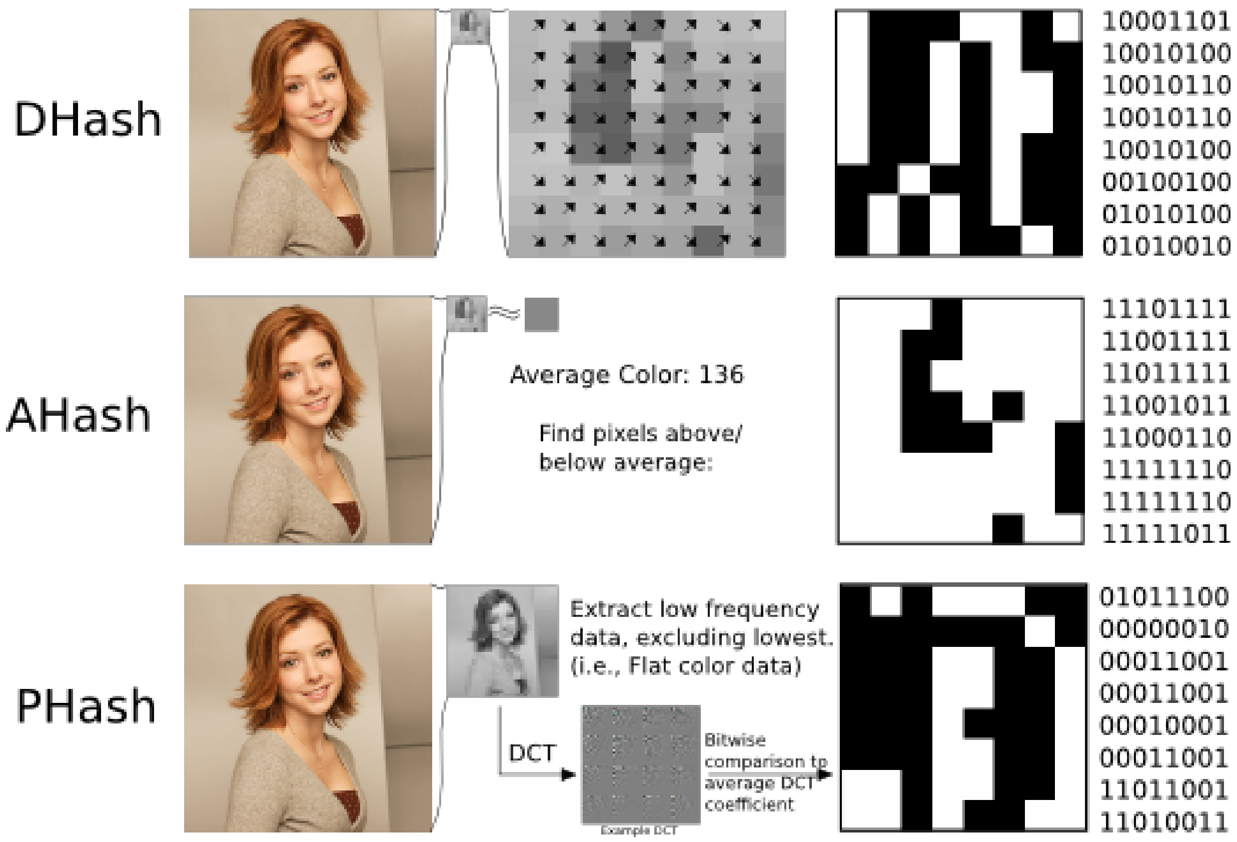


Figure A.1: Visual Comparison of Hashing Algorithms