# Rasterized Image Databases with LSH for Compression, Search and Duplicate Detection

Zoya Bylinskii
MIT CSAIL

Maria Shugrina
MIT CSAIL

Andrew Spielberg
MIT CSAIL

Wei Zhao
MIT CSAIL

## ABSTRACT

TODO.

## 1. INTRODUCTION

Large collections of images are ubiquitous in the modern digital world. According to one 2014 Internet Trends report, more than 1.8 billion images are uploaded to the internet every day [3]. Our work is inspired by the intuition that there must be a lot of redundancy in large image collections, and that this redundancy could be exploited for more efficient storage and for applications such as duplicate detection.

We focus on image redundancy on the patch level, assuming that large collections of images must have many patches which are nearly the same. Our goal is to store a set of images as a database of similar patches, where similar patches may be shared between images, such that we minimize the storage space while maintaining certain *quality* of reconstructed images. In effect, this results in lossy compression. More concretely, we aim to choose patch similarity criterion, and search and reconstruction algorithms such that:

- the database size is smaller than if full images were stored
- the images can be reconstructed from the database in real time
- the reconstructed images fulfill certain quality requirements (See Sec. 4.1)

These conflicting introduce a number of tradeoffs, such as size of the database versus image quality. The goal of this paper is as much to produce a working system as to build up the analytical foundations that allow making these tradeoffs.

In Section 3, we explain our method of “patchifying” images, storing them in a database, and reconstructing stored images. We also discuss image similarity, hashing schemes and indices in this core section. In Section 4, we provide analytical groundwork for selecting optimal image patch sizes, similarity threshold, and the quality metrics appropriate for evaluation. Finally, in Section 5 we evaluate our system on real data, using analytical tools from the previous section. In addition, in Section 6, we briefly touch on applications that come naturally from a patch database, including similar image retrieval and duplicate detection.

## 2. RELATED WORK

Image databases, in particular rasterized.
Image compression, in particular JPEGs.
Image similarity functions.
Image quality functions.
Hashing, in particular [1].

## 3. METHOD

To limit the complexity, we assume each image to be a square of $m^2$ pixels, and for all patches to be squares of the same size. We formally describe our method in Sec. 3.1. To summarize, we first segment each image into patches and store them in the `patches` database. Only patches sufficiently different from all the patches in `patches` are stored (See Sec. 3.2). Instead of storing each image explicitly, we approximate it by storing pointers to the patches approximating its original patches. In Sec. 3.3, we describe the search algorithm used to quickly retrieve similar patches, and in Sec. 3.4 we detail the indices that make image reconstruction faster. Finally, in Sec. 3.5 we provide implementation details.

### 3.1 Overview

Our method is governed by the following parameters:

- $n$ - width and height of all patches[1]
- $S$ - similarity function from two $n \times n$ images to $\mathbb{R}$
- $T$ - similarity threshold

The $n \times n$ patches are stored as byte data in the `patches` table. To allow image reconstruction, we also store $(m/n)^2$ patch pointers for each image in the `patch_pointers` table. The full schema looks as follows:

```
patches(id int PRIMARY KEY,
        patch bytea);

images(imgid int PRIMARY KEY);
```

[1]We assume that $m \mod n$ is 0.

```
patch_pointers(imgid int REFERENCES images(imgid),
               patch_id int REFERENCES patches(id),
               x int,
               y int);
```

where `patch_pointers.x` and `patch_pointers.y` refer to the left top corner location of each patch in the image.

Given these tables, inserting an image into the database proceeds as follows:

**Algorithm 1** Insert Image $I$ into database

```
1: $Patches \leftarrow$ CutIntoPatches(I, patch_size=$n$)
2: for $P$ in $Patches$ do
3:     $SimPat \leftarrow$FindLikelySimilarPatches($P$)
4:     $P_{closest} \leftarrow argmin\{S(P, P_i)\}$
5:     if then$S(P, P_i) > T$
6:         insert $P$ into patches
```

## 3.2 Patch Similarity

There are many image similarity metrics that have been developed for images (See [4] for a good survey.), and our method is applicable to any metric that involves Euclidean distance over image features, its stacked RGB pixel values in the simplest case.

For the purpose of this project, we choose to use squared Euclidean distance over (CIE)LUV color space, which is known to be more perceptually uniform than the standard RGB color space [2]. Given two $n \times n$ patches $P_1$ and $P_2$, we stack $3 \times n^2$ LUV pixel values into vectors $p_1$ and $p_2$, and evaluate similarity $S$ as follows:

$$S(P_1, P_2) = \frac{||p_1 - p_2||^2}{n^2}$$

where $||\cdot||$ denotes standard Euclidean norm. We normalize by the dimensionality of the space to allow us to keep the similarity threshold $T$ independent of the patch size.

The use of Euclidean distance for patch similarity allows us to use Locality Sensitive Hashing to retreive patches that are likely to be similar, as detailed in the next section.

## 3.3 LSH for Patches

## 3.4 Image Reconstruction

In order to reconstruct images quickly, we

## 3.5 Implementation

We used *postgresql* to construct our database, and used Java API to talk to the database from a custom executable. Locality sensitive hashing, image segmentation and reconstruction were all implemented in Java, and used to construct a hash table on patches in *postbresql*.

Our code is available at:

`https://github.com/shumash/db_project`

## 4. ANALYSIS

## 4.1 Quality Metrics

There are many possible formulations of these quality constraints; we detail those that we considered for this paper in section TODO. Beyond mathematical metrics, subjective methods are also interesting to consider for formulating the quality constraints; one could imagine a scenario in which image quality is assessed by humans through a crowdsourced system, perhaps using an engine such as Amazon's Mechanical Turk TODO: cite. We consider such subjective similarity metrics outside the scope of this paper and focus on the mathematical metrics for now.

## 4.2 Patch Size

TODO(Zoya,Andrew): tradeoffs for patch size

Assume for now that we choose to store $p$ patches in our auxiliary table. In practice, we choose $p$ to be a function $p$: $Function \rightarrow \mathbb{N}$ which maps from our similarity metric to a number of patches to store. Assume also that each patch is square and composed of $n^2$ pixels, where $n$ is a user defined parameter. We further assume that each pixel requires 8 bytes to store and that each pointer is 8 bytes (a standard integer for a 64-bit system). Under this "image only" scheme, in the case where we have $i$ images, the cost $c_i$ to store all the images in our database is:

$$c_i(i, m) = 8im^2 \tag{1}$$

In the case where we store pointers to patches, we have two tables: one table to store pointers to image patch exemplars, and a second table to store the exemplar data themselves. Under this "patch pointer"scheme, in the case where we have $i$ images and $p$ patches, the cost $c_p$ to store all the images in our database is:

$$c_p(i, p, m, n) = 8i(\frac{m}{n})^2 + 8pn^2 \tag{2}$$

.

The first term is the cost of storing the pointer data, while the second term is the cost of storing the patch exemplars themselves.

## 4.3 Similarity Threshold

TODO: tradeoffs for similarity threshold

## 5. PERFORMANCE

In this section we evaluate the performance of the implementation of our method on a real collection of X images of Y size.

## 5.1 Quantitative

## 5.2 Qualitative

## 6. APPLICATIONS

One of the appeals of this approximate patch-based approach is that it naturally lends itself to applications. In this section we describe methods to use our database for two applications - duplicate detection (Sec. 6.1) and similar image retrieval (Sec. 6.2).

## 6.1 Duplicate Detection

## 6.2 Similar Image Retrieval

## 7. CONCLUSION

TODO.

## 8. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, Jan. 2008.

[2] H. Kekre and V. K. Banura. Performance comparison of image retrieval using kfcg with assorted pixel window sizes in rgb and luv color spaces 1. 2012.

[3] M. Meeker. Internet trends 2014 - code conference, 2014.

[4] M. Yasmin, M. Sharif, and S. Mohsin. Use of low level features for content based image retrieval: Survey. *Research Journal of Recent Sciences*, 2277:2502, 2013.