ABSTRACT

We present a novel strategy for patch-based lossy compression which exploits the inevitable redundancy found in large image collections. We describe a scalable PostgreSQL-based implementation leveraging the indexing infrastructure found in relational database management systems (DBMS). We decompose images into sets of *patches* which we define to be small contiguous subregions of images. Since many image textures (e.g. sky, ocean, forest, walls, etc.) are ubiquitous in image datasets, rather than store all the information of these patches for every image, we only store pointers to approximating patches in a *patch dictionary*. This dictionary is grown in an online fashion as new images are added. New patches are added to the dictionary when no similar-enough patch can currently be found in the database under our chosen patch distance function. In order to efficiently retrieve patches from our database, we make use of locality-sensitive hashing and a number of key optimizations. We analytically compute the savings of our compression scheme, and experimentally demonstrate its performance over various image categories of the large SUN 2012 database. Finally, we provide subjective and quantitative evaluations of our compression quality.

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 [**?**]. 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 a patch distance criterion, matching 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. 6)

These goals 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.

Section 2 covers some related work on image compression, raster databases, and patch-based computer vision applications. In section 3, we explain our method of "patchifying" images, storing them in a database, and reconstructing stored images. In section 4 we discuss how we accomplish the fast retrieval of similar patches via locality sensitive hashing (LSH), which is crucial to the feasibility of our system. In section 5 we discuss other important database optimizations required for performance. In section 6, we provide the analytical groundwork for selecting optimal image patch sizes, distance thresholds, and quality metrics appropriate for evaluation. Finally, in sections 7 and 8, we quantitatively and qualitatively evaluate our full working system on real data, using the analytical tools developed from the previous section. In addition, in section 9, we briefly touch on applications that derive naturally from a patch database, including similar image retrieval and duplicate detection, as well as a fun photomosaic application. We conclude by discussing future extensions.

2. RELATED WORK

Lossless image compression techniques like *area coding* or *Huffman coding* give quality guarantees but do not always give sufficient savings (especially for unstructured images). Lossy image compression algorithms, on the other hand, are quite popular and used for most applications dealing with natural images, including storage in image datasets, and web transmission. Often, lossy compression depends on a quantization of image regions (e.g. all pixels in a block of the image receive the same color value). For instance, the *JPEG compression* standard partitions an image into non overlapping blocks and applies a discrete cosine transform to each block, quantizing the resulting coefficients. *Fractal compression* techniques [**?**] apply an iterative algorithm to images to encode images as fractal codes, capable of representing different parts of the images at different levels of detail. This algorithm makes use of self-similarities and is computationally expensive. However, due to fast decoding, it can be used for file downloads. All of these approaches, however, operate on a *within-image* basis, compressing an image using either a pre-specified dictionary (quantization) or using within-image similarity. Thus, these approaches provide constant savings for image databases of arbitrary sizes.

In the age of big data, when the amount of images in image collections grows at enormous rates, a compression scheme that scales with the database size seems most appropriate. Thus, we are interested in compression schemes that take into account *across-image* redundancy, not just *within-image* redundancy.

Some of our inspiration comes from *raster databases*, which encode images (often of geospatial data) as a set of smaller images/regions with locations in the original large image. Such a set-up is convenient for transmission or data loading, as it is possible to load and process only parts of the image at a time. Geodatabases, such as *ArcGIS* are set-up this way [**?**].

We combine the idea of raster databases with the idea of image compression. Patch-based image representations are also used in computer vision for various tasks, including image matching [**?**], object recognition [**?**], and image processing [**?**]. Considering images as collections of patches allows for matching without rigid spatial constraints. In other words, such approaches can often find approximate image matches, such as different viewpoints of the same scene. Patch-based approaches have thus proven to be sufficient for scene recognition applications. We adapt a patch-based approach as well. Even very coarse-grained patches can provide sufficient visual information for scene recognition purposes. Thus, if we reconstruct images from approximately-similar coarse-grained patches, they will still provide sufficient input to scene classification algorithms, for instance.

3. APPROACH

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, n^2 pixels. We formally describe our method in Sec. 3.1. To summarize, we first segment each image into patches and store them in the `patch_dict` table, a dictionary of patches. Only patches sufficiently different from all the other patches in `patch_dict` are subsequently added to the dictionary (see sec. 3.2). We describe our patch distance in sec. 3.3 and our implementation details in 3.4. Instead of storing each image explicitly, we store pointers to the patches approximating its original patches. Thus, during reconstruction, we simply stitch together the patches using pointers (sec. 3.5).

3.1 Overview

Our method is governed by the following parameters:

- m - The width and height of all images.
- n - The width and height of all patches[1].
- k - The number of images in our database.
- S - A distance function $S\colon \mathbb{I}_{n\times n} \times \mathbb{I}_{n\times n} \to \mathbb{N}$, where we define $\mathbb{I}_{n\times n}$ to be the space of all $n \times n$ image patches. Section 3.3 details distance measures. The distance function should be at least a pseudometric, so $S(P_1, P_2) = 0$ if and only if patches P_1 and P_2 are the same, and so that the triangle inequality holds.
- T - A distance threshold, $T \in \mathbb{R}$; used as a maximum value we allow on S for patch mappings.

It is worth noting that we propose a *lossy* compression scheme ($T > 0$). For the remainder of the paper, when we use the term *images* we are referring to entire images from our database. When we use the term *image patches* we are referring to small $n \times n$ contiguous portions of images in our database. When we use the term *dictionary patches*, we are specifically referring to patches which we have chosen to store in our `patch_dict` and use for compression. Here we present an overview of our compression method; in subsequent subsections we delve into the details.

3.2 Algorithm

We begin our compression algorithm by seeding `patch_dict` with an initial set of dictionary patches. These patches are chosen from randomly selected $n \times n$ image patches from the entire image database (see sec. 6.2 for a discussion of this seeding strategy). During the image insertion step, we partition each of our images into $\left(\frac{m}{n}\right)^2$ non-overlapping patches, with the intent of mapping each image patch P_j to a patch in `patch_dict`. Thus, rather than storing the original image patch for a given image, we simply store a pointer to a patch in `patch_dict`. The dictionary patch we choose is the one which is closest to the image patch according to some distance metric S, i.e. the patch P_{NN} in `patch_dict` such that $S(P_{NN}, P_j)$ is minimized. If $S(P_{NN}, P_j) > T$, we then store P_j as a new patch in `patch_dict` and add a pointer from the image to this dictionary patch (at the corresponding (x, y) location in the image). Algorithm 1 summarizes the image insertion procedure. Assuming that our patch dictionary is a good sample of the image patches in our image database, adding additional patches should be a relatively rare procedure. We discuss how often these extra insertions are needed in sec. 6.5. Thus, the space savings come from only needing to store an effective pointer for each image patch, rather than the entire patch data. Note that the maximum threshold on the distance of image patches and dictionary patches guarantees that each compressed image is at most $\frac{mT}{n}$ away from its original counterpart in S.

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

Algorithm 1 Basic alg. to insert image I into database

```
Patches ← Patchify(I, n)
for P_j in Patches do
    P_NN ← argmin_{P_i ∈ patch_dict}{S(P_i, P_j)}
    if S(P_NN, P_j) > T then
        insert P_j into patches
```

With a large table of patches, finding the closest patch can be computationally expensive. In order to speed up the search, we employ *locality sensitive hashing* (LSH). Although this softens the constraint that we always find the closest dictionary patch in `patch_dict` for each image patch, the closest patch is still found with very high probability, and in expectation the selected patch is still very similar. Section 5 details nearest-neighbor retrieval, and alg. 2 includes the algorithm updated to account for this optimization.

Algorithm 2 Modification of alg. 1 with approximation

```
Patches ← Patchify(I, n)
for P_j in Patches do
    SimPat ← FindLikelySimilarPatches(P_j, patch_dict)
    P_ANN ← argmin_{P_i ∈ SimPat}{S(P_i, P_j)}
    if S(P_ANN, P_j) > T then
        insert P_j into patches
```

We will define $M\colon \mathbb{I}_{n\times n} \to \mathbb{I}_{n\times n}$ to return the closest patch to P_j from the set of patches returned by FindLikelySimilarPatches. $M(P_j)$ is the approximate nearest neighbor to P_j. Thus, our compression problem can formally be stated as choosing a selection of image patch to dictionary patch mappings which minimizes the storage space usage of our patch table, while constraining each image tile to be at most T away from its mapped patch. In other words,

$$\begin{aligned} \underset{patch\_dict, M}{\text{minimize}} \quad & c(k, d, m, n) \\ \text{subject to} \quad & S(P_j, M(P_j)) \le T,\ j = 1, \ldots, k\left(\frac{m}{n}\right)^2. \end{aligned}$$

where $c(\cdot, \cdot, \cdot, \cdot)$ is a cost function as defined in section 6.1.1, d is the number of patches in the dictionary (i.e. $d = |patch\_dict|$), and k, m, n are as defined in 3.1.

Given our pointer representation, we are able to construct the compressed image quite efficiently. Given an image identifier, we iterate over all patch pointers stored with it, associated with each image location (x, y), and simply insert the pointed to patch at that location.

3.3 Patch Distance Metric

There are many image similarity/distance metrics that have been developed for images (see [**?**] for a good survey),

and our method is applicable to any metric that involves Euclidean distance over image features, its stacked color channel pixel values being the simplest case.

For the purpose of this project, we choose to use squared Euclidean distance over (CIE)LUV color space. Given two $n \times n$ patches P_i and P_j, we evaluate distance S per color channel u as follows:

$$S(P_i, P_j, u) = \frac{||P_i(u) - P_j(u)||^2}{n^2}$$

where $||\cdot||$ denotes standard Euclidean norm. We normalize by the dimensionality of the space to allow us to keep the distance threshold independent of the patch size. See section 6.3 for more details. A benefit of using a Euclidean distance metric is that it allows us to use LSH to retrieve patches that are likely to be similar.

3.4 Implementation

We used *PostgreSQL* to construct our database, and used the 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 *PostgreSQL*.

Our code is available at:

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

The $n \times n$ patches are stored as byte data in the `patch_dict` table. We store the patch pointers for each image in the `patch_pointers` table. The full schema looks as follows:

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

images(id int PRIMARY KEY);

patch_pointers(img_id int REFERENCES images(id),
               patch_id int REFERENCES patch_dict(id),
               x int,
               y int);

patch_hashes(
   patch_id int PRIMARY KEY REFERENCES patch_dict(id),
   hash int);
```

where `patch_pointers.x` and `patch_pointers.y` refer to the left top corner location of each patch in the image.
A visualization of our schema is provided in fig. 1.

3.5 Image Reconstruction

To reconstruct an image, we simply follow the (logical) pointers in the corresponding `patch_pointers` database table entry to retrieve the patches from the `patch_dict` for each (x, y) location in the image:

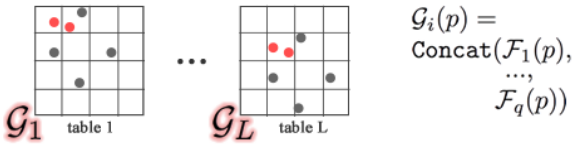
Algorithm 3 Image reconstruction

1: $PatchPointers \leftarrow$ `getAllPatchPointers`$(patch\_pointers)$
2: $PatchData \leftarrow$ `getAllPatches`$(patch\_dict, PatchPointers)$
3: $ResultImage \leftarrow []$
4: **for** P_d in $PatchData$ **do**
5: $\quad ResultImage.set(P_d.x, P_d.y, P_d.patch)$

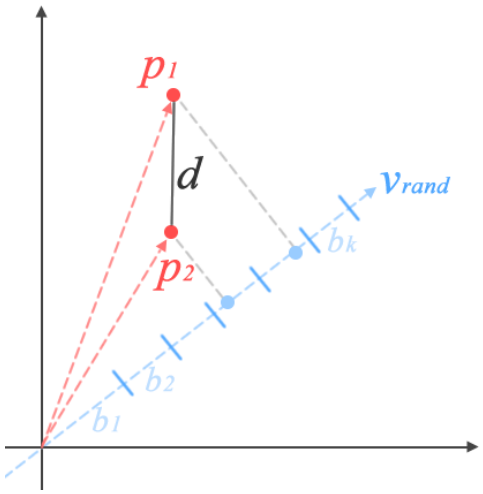
4. NEAR NEIGHBOR SEARCH

Fast retrieval of similar patches is crucial for making construction of a sizable patch-based database feasible. This is essentially near-neighbor retrieval in relatively high dimensions $3 \cdot n^2$ (1875 for patch size $n = 25$). Image retrieval has been addressed in a number of papers, including more complex feature representations [?]. Our problem is somewhat different from most of previous work in that the variability in small patches is much less than in regular-sized images, semantic information is irrelevant, and vectors are much shorter than for regular-sized images. Thus, we focus on tuning a simple Locality-Sensitive Hashing variant for our particular application.

4.1 Locality-Sensitive Hashing



(a) Typical LSH uses several tables.



(b) Random projection hashing family $\mathcal{F}$.

Figure 2: a) LSH typically constructs L hash tables, each relying on an amplified hash function $\mathcal{G}$, composed of a concatenation of simpler hash functions $\mathcal{F}$. b) In random projection hashing, each $\mathcal{F}$ is evaluated by projecting a point p (a patch color vector in our case) on a randomly chosen unit vector, and binning it into uniform bins.

Locality-Sensitive Hashing (LSH) [?] is a popular approach for approximate near-neighbor search in high dimensions. The high-level idea behind LSH is using a set of hashing functions that map a vector P_i to its bin $b(P_i)$ such that:

$$P[b(P_i) = b(P_j) | S(P_i, P_j) < T] > P_1$$
$$P[b(P_i) = b(P_j) | S(P_i, P_j) > cT] < P_2$$

where the collision probability for vectors that are close together is high ($> P_1$), and the probability of collision for vectors that are further apart is low ($< P_2$).

To achieve high guarantees on finding the nearest neighbor, LSH typically requires multiple hash tables. For example, two red points in Fig. 2 are close, but they fall in

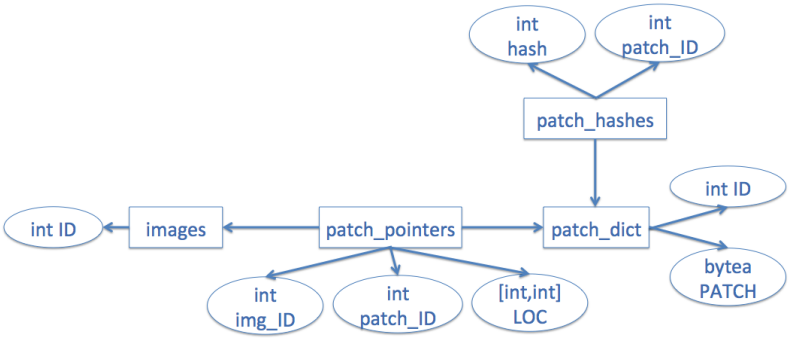


Figure 1: The proposed database schema.

different bins in table 1. However, in some table (say Table L), they are likely to end up in the same bin. This amounts to taking an OR over collisions in all the tables, amplifying probabilities. Implementing this scenario in the context of a database incurs a significant storage overhead, as for each patch P_i we not only need to store the patch itself, but also L hash indices. By the same token, queries must be made to all the hash tables, which incurs significant overhead at near neighbor search time.

High precision near neighbor search is not necessary for our application, and we focus, instead, on *efficiency*. In order to accomplish that we aim to **minimize the probability of collision for dis-similar patches**, as long as probability of collision for similar patches results in finding enough neighbors for a suitable compression ratio. To this end, we aim to answer the question: *how well can the system perform with just one hash table?*

In order to optimize the performance of our single hash table, we want to *amplify* the probability of a single hash function from a family $\mathcal{F}$ by using an AND operator on collisions with several hash functions $\mathcal{F}_1 ... \mathcal{F}_q$ from that family. This is a standard technique which results in an amplified hash function $\mathcal{G}$:

$$\mathcal{G}(P_i) = \texttt{ConcatBits}(\mathcal{F}_1(P_i), ..., \mathcal{F}_q(P_i)) \quad (1)$$

where `ConcatBits` simply allocates a constant number of bits to each hash and concatenates them into a single value. The probability that dis-similar patches collide with this amplified function is $(P_2)^q$, where P_2 is the probability of collision using just one function in the family. We formalize our Near Neighbor search in Sec. 4.2, and take a closer look at hashing functions in the following sections. In Sec. 4.3, we introduce a common choice for a hash function family $\mathcal{F}$, and optimize it for our application in Sec. 4.4 and 4.5.

4.2 Near-Neighbor Search with LSH

Using a single amplified hash function $\mathcal{G}$, finding all likely neighbors of a patch simply amounts to computing its hash value and taking all the patches that fall into the same hash bin. More formally, we define our approach in alg. 2 as follows:

`FindLikelySimilarPatches`(P_j):
1: $h \leftarrow \mathcal{G}$ (`ToVector`(P_j))
2: $SimPat \leftarrow$ `select patch from patch_dict where id in (select patch_id from patch_hashes where hash = h)`

Of course, the quality of the result depends heavily on the properties of the hash function, which is something we discuss next.

4.3 Random Projections Hashing

A common choice for a hash function family $\mathcal{F}$ is the family of random projection functions. In this scheme, a given $\mathcal{F}_i(P_j) \in \mathcal{F}$ is evaluated by projecting P_j onto a randomly chosen unit vector v_i, and binning the value into a randomly chosen bin (see fig. 2, where bins are uniform across dot product values:

$$\mathcal{F}_i(P_j) = \lfloor \frac{P_i \cdot \mathbf{v}_i + b}{w} \rfloor \quad (2)$$

We choose to work with this approach, but operate under the constraint that the value of the amplified $\mathcal{G}$ of q functions $\mathcal{F}_i \in \mathcal{F}$ must fit in a 32-bit int. This only affords us 10 hashes in our amplification, with a bin size of 8 (3 bits).

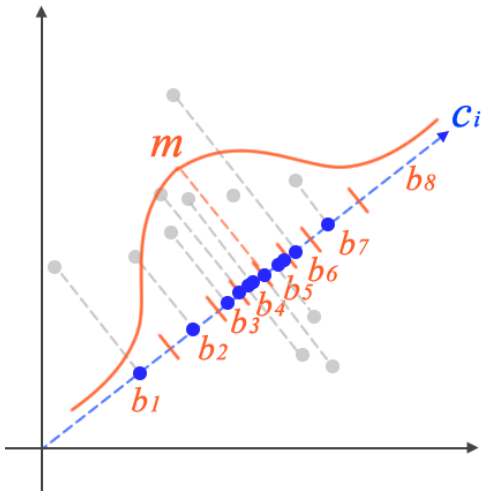
It is unlikely that the projection vector picked randomly will have desirable properties, given high dimensions. For example, it is quite likely that typical patches will be orthogonal to most vectors picked, causing many dis-similar patches to end up in the same bin (and this is exactly the behavior we observed in our results).

4.4 PCA-based Projection Hashing

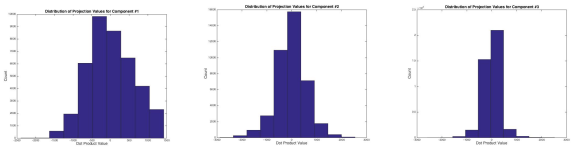
To ensure that patches are well-distributed among the bins, avoiding database lookup of many potential neighbors, we propose a *PCA-based hashing scheme*. Under this scheme, we run Principal Component Analysis (PCA) to compute directions of largest variance for typical image patches and use the 10 first principal components as the projection directions (see fig. 3). This ensures that the projections of typical patches are as far apart as possible. To further optimize the binning, we project a set of patches (grey dots in fig. 3) onto each principal component $\mathbf{v}_i$ (blue dots in fig. 3), and approximate the distribution of the dot product values by a Gaussian $\mathcal{N}_i(\mu_i, \sigma_i)$ (orange curve in fig. 3). In order to bin a dot product value into one of 8 bins (3 bits/bin), we adapt the bin size to μ_i and σ_i.

In our experimental setup we uniformly sampled patches from two distinct sets of 1000 images from the SUN [**?**] database (and distinct from the set of images we inserted into the database), and used the patches to:

- compute PCA vectors in Luv color scheme on the `train set` of 80,000 25x25 patches (see fig. 4)



(a) PCA-based hashing



(b) Dot product distributions

Figure 3: PCA-based hashing uses directions of maximal variance in patches as projection vectors, and adapt the bin size for each projection vector using data. In b) we show distributions of patch vector projections on the 3 principal components (magnitude of dot product).

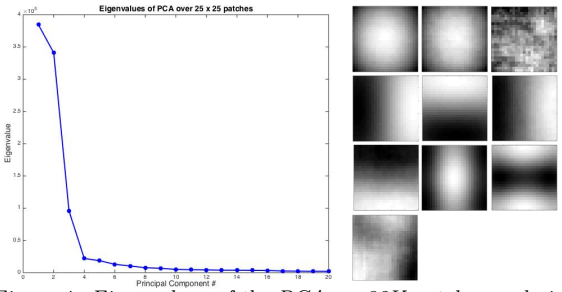


Figure 4: Eigenvalues of the PCA on 80K patches and visualization of the L-channel (lightness) of the first 10 components.

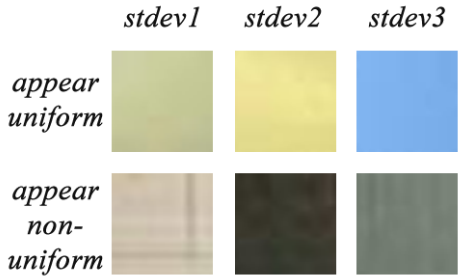
- compute distribution of typical projections on 10 first principal components using the `dev set` of 40,000 25x25 patches (see fig. 3(b)).

We found that this choice of a hashing function has *a significant positive effect on performance* (see sec. 7).

4.5 Hashing Uniform Patches

The difficulty of Near Neighbor search for images arises from their high dimensionality. However, natural images contain many uniform or nearly uniform patches, which can be easily indexed using their quantized color. We experimented with using a different hashing scheme for patches that are nearly uniform, quantizing them into fewer than 1000 bins by the mean color.

To determine if patch P is uniformly colored, we first looked at the standard deviation u_2 of the patch in all channels, which is just the L2-norm of $P - \mu(P)$, where μP is a patch with every pixel set to the mean of P. However, we found that the L2-norm is insensitive to outliers (which are very apparent in color patches), and patches with similar u_2 values could appear both uniform and nonuniform (see fig. 5(a)).



(a) Pitfalls of standard deviation

(b) Uniformity classification with p-norm

Figure 5: Standard deviation in a) is not very sensitive to outliers, as patches with nearly identical values of stdev1, stdev2 and stdev3 may appear either uniform or non-uniform, as shown in the table. In b), our classification of patches using p-norm anecdotally shows more reliable results.

Instead, we decided to classify uniformity using u_4, an L4-norm of $P - \mu(P)$, split into channels. Higher-order forms tend to collapse all values < 1.0 to zero, and expand all values > 1.0 to infinity. Hence, we normalize the elements of $P - \mu(P)$ by our channel threshold T to ensure this property. The best approach would be to pick uniform thresholds on u_4 using a training set, but in the interest of time we hand-picked a threshold of 0.1 for all the channels. See fig. 5(b) for results using this classification on a random set of patches.

We evaluate the performance of PCA-based hashing compared to our hybrid PCA+uniform patch hashing in Sec. 7.

5. DATABASE OPTIMIZATION

In this section we discuss additional optimizations to the database to make efficient construction of patch database feasible. Here and below I_{new} is an image about to be inserted into the database, and $P_1^n ... P_j^n ...$ are patches comprising it. To make large databases practical, our objective is to make the insertion process as fast as possible, given any of the Near Neighbor search methods described in Sec. 4.

5.1 Database Queries

Referring back to Alg. 2 and the definition of `FindLikelySimilarPatches` in Sec. 4.2, it is clear that to insert I_{new}, about $2\left(\frac{m}{n}\right)^2$ database queries are issued: $\left(\frac{m}{n}\right)^2$ for finding near neighbors and at most $\left(\frac{m}{n}\right)^2$ for inserting any patches without matches. This incurrs a significant overhead, and we have modified our algorithm to make only at most 2 queries for every new image insertion. To accomplish this, we use batch query and insert, which result in 2 database queries, once to find all the likely similar patches, and once to insert new patches (See Alg. 4).

Algorithm 4 Optimization of alg. 2 for DB Queries

```
Patches ← Patchify(I, n)
Hashes ← [ ]
for P_j in Patches do
    P_j.h ← G (ToVector(P_j))
    add P_j.h to Hashes

NewPatchesToStore ← [ ]
StoredPatches ← HashMap: hash → [stored patches]
StoredPatches.FillFrom( select hash, patch from
patch_dict, patch_hashes where hash in Hashes)
for P_j in Patches do
    SimPat ← StoredPatches[P_j.h]
    P_ANN ← argmin_{P_i ∈ SimPat}{S(P_i, P_j)}
    if S(P_ANN, P_j) > T then
        NewPatchesToStore.Add(P_j)
        StoredPatches[P_j.h].Add(P_j)
BatchInsert(NewPatchesToStore)
```

Please note that this hides some of the complexity, as we also need to keep track of pointer data, i.e. which stored patch ID each tile in the new image should point to. If `HashMap` contains newly processed patches that have not yet been inserted into the database, we need to make sure that in the end the pointers for the image contain the right database IDs.

5.2 Patch Buffer Pool

In order to further optimize performance, we implemented a `BufferPool` for patches with a least-recently-used (LRU) eviction policy. The LRU policy was picked based on the intuition that similar images are often processed together. This is particularly true if the database is constructed sequentially from a categorized database, such as SUN [**?**].

The function of the `BufferPool` is two-fold. In addition to minimizing database queries and random I/O for reading patches from disk, the `BufferPool` stores the hash value of each patch, as well as its vector form for faster computation of the distance S. These values are computed in a lazy fashion - only when requested. Given our unoptimized Java implementation of image vectorization and dot product, we found these measures to yield a non-negligible speed-up.

6. PARAMETER ESTIMATION

A number of parameters can be tweaked to change the patch matching and storage, and different choices may be appropriate for different applications and performance requirements (both quantitative and qualitative). These parameters include the size of the input images, the size of the patches extracted, the sampling strategy used to seed the dictionary, the distance metric and thresholds used to compare patches, as well as the parameters required for indexing and retrieving patch matches (discussed in Sec. 4). Here we discuss some of the parameter choices made and the experiments that lead up to these choices. Other possible choices are discussed in Sec.9.2.

Our quantitative performance metrics involve examining how the patch dictionary size grows with the addition of new images to the database (the growth function and rate) and the compression ratio per image (viewed as a distribution over compression ratios and summarized as the average compression ratio). Qualitative evaluations involve determining whether a human can spot compression artifacts and how salient they are in the images. The authors of this paper manually examined images reconstructed from the dictionary patches. A crowdsourced evaluation strategy would be more appropriate for larger-scale studies, but is beyond the scope of this paper.

There will always be a trade-off between compression benefits (storage: patch dictionary size and speed: image reconstruction time), and reconstruction quality. For many computer vision tasks including scene recognition (and thus retrieval), imperfect reconstructions with artifacts may not be a problem as long as the overall scene structure does not change. For instance, [**?**] has shown that with images of pixel dimension 32x32, humans are already able to achieve over 80% scene and object recognition rate. See fig.6 for a demonstration of an image that has serious reconstruction artifacts, but when downsampled to a thumbnail, they become insignificant, and thus may have no impact on visual recognition.

6.1 Patch Size

Larger patches contain more image structure, and thus the probability that another patch contains the same or similar image content decreases with the number of pixels in a patch. At larger granularities it becomes increasingly harder to find matching patches in the patch dictionary, and the closest matching patches for textured regions might introduce artifacts (see fig.7). At the same time, patches that are too small do not offer as efficient a compression strategy. We must balance the costs of storing pointers to patches for each image in our database, as well as all the patches themselves, against the costs of storing the images in their original form. This calculation is investigated further below.

6.1.1 Cost Evaluation

Assume, as before, that d is the number of patches in `patch_dict`. In practice, d is a function of the number of images added to the database as well as the distance function S and threshold T. We further assume that each pixel requires 3 bytes to store and that each pointer is 8 bytes (a standard integer for a 64-bit system). Then, with k images

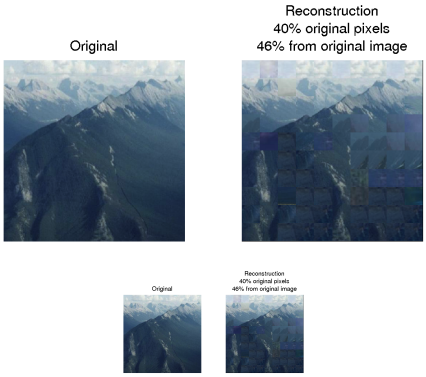


Figure 6: For demonstration purposes only, we choose a large patch size and low distance threshold. Under these parameters, the original image is reconstructed to take up only 40% of its original size (in pixels). The 60% of the patches that have been replaced come either from the same image (46% of them), or from other images (the remaining 64%). Notice that when the size of the image and its reconstruction are halved, the artifacts already become visually insignificant, and would not impair a scene recognition or search task.

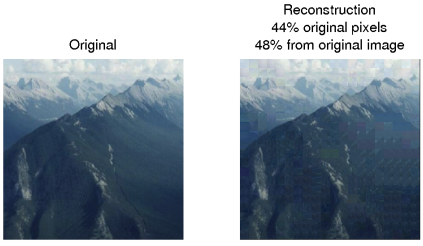


Figure 7: Compare this image reconstruction, computed with a dictionary of 25×25 pixel patches with the reconstruction in fig.6, computed with 50×50 patches. In both cases, a similar threshold is used (scaled to the patch size, as discussed in sec. 6.3) but the visual artifacts are less noticeable because smaller patches have less contained structure, and are more likely to be homogenous in appearance.

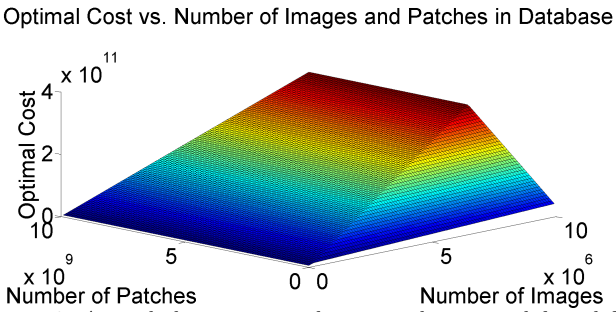


Figure 8: A graph demonstrating how c_{opt} changes with k and d for $m = 100$ and $n = 10$. Note the line of discontinuity where $d = 357.3k$ - this is the line where the costs of c and c' intersect.

in the database, the full cost to store all the original images (no patch-based compression scheme) in our database is:

$$c'(k, m) = 3km^2 \tag{3}$$

In the case where we store pointers to patches, we have two tables: one table to store pointers to dictionary patches, and a second table to store the dictionary patches themselves. Under this "patch pointer" scheme, with k images and d patches, the cost c to store all the images in our database is:

$$c(k, d, m, n) = 4k\left(\frac{m}{n}\right)^2 + d(4 + 3n^2) \tag{4}$$

.

The first term is the cost of storing the pointer data, while the second term is the cost of storing the patches themselves. The 4 constant comes from the fact that each pointer is a 4-bit int, while each patch requires storing 3 color channels for n^2 pixels, as well as a 4 byte hash value. Note that when dealing with extremely large image collections, it is possible to have over 2^{32} patches; in this case a `bigint` would be required for storing pointers, changing the 4 to a 8 and slightly changing the subsequent analysis. In practice, we also store a bit more metadata to make managing the database easier, so this analysis should be thought of more as an optimal bound on storage.

Given these two equations, for a fixed m and n, we can easily see that our compressive scheme becomes more space-efficient than storing the original images when:

$$d < \frac{m^2(3 - 4/(n^2))k}{4 + 3n^2} \tag{5}$$

As long as we choose a distance threshold such that new image patches get added at a rate that guarantees this inequality is satisfied, our compressive method of image storage will save space. Figure 8 shows an example of how the optimal storage cost changes with different patch and image counts, where the optimal cost is defined as $c_{opt} = \min(c, c')$; in other words, storing the images using the less expensive method. See fig. 8.

6.2 Sampling strategies

A patch dictionary can be built up incrementally, adding new patches as new images are added to the database. A potential problem with this approach is that image reconstruction quality will tend to decrease with the order in which

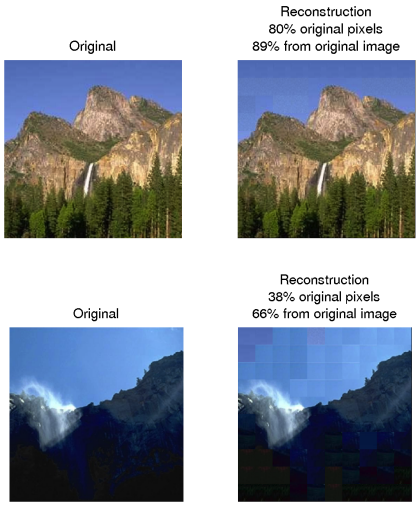


Figure 9: Example of a biased patch dictionary construction strategy, leading to non-uniformity in image reconstruction quality. Images added to the database earlier (top row) are better reconstructed (due to more patch samples in the database) than images added later (bottom row), constrained to be constructed out of patches added initially. The sky pixels in the image added later are borrowed from sky pixels of other images (44% of the pixels in this image come from other images, compared to only 11% in the image on the first row). Note: here we use a very high patch distance threshold and large patch size for demonstration purposes only, to emphasize the artifacts created.

images are added, such that images added to the database earlier will tend to have more patches that correspond to them (see Fig.9 for an example). A strategy with a more even distribution of reconstruction quality over images involves starting with a batch of images, and seeding the dictionary by randomly sampling patches from a set of images from the batch. This is the strategy we employ.

6.3 Distance Function

Many image (more specifically, patch) distance functions are possible, each with its own distinct set of parameters that can be tweaked for the required application. Because we are dealing with patches of a size specifically chosen to increase within-patch homogeneity, we do not consider cases of patches containing objects (the most we expect is an object boundary or simple texture), and thus do not need to consider complex image similarity/distance functions (involving SIFT, GIST, and other computer vision features). We can constrain ourselves to color distance, and split a patch P_i into 3 LUV color channels: $P_i(L), P_i(U), P_i(V)$.

Then we consider two patches P_i and P_j similar when, given $n \times n$ pixels, all of the following are true:

$$\frac{1}{n^2}||P_i(L) - P_j(L)||^2 < T_1$$
$$\frac{1}{n^2}||P_i(U) - P_j(U)||^2 < T_2$$
$$\frac{1}{n^2}||P_i(V) - P_j(V)||^2 < T_3$$

The $\frac{1}{n^2}$ term allows us to normalize for patch size, so that the threshold values chosen becomes independent of patch size. Here we constrain the average distance value of all the pixels in a patch to fit a threshold, whereas it is possible to have alternative constraints (where instead of the average, the maximal pixel difference or the variance of the pixel differences or some other measure over pixels in a patch, is compared to a threshold).

Note additionally that if instead we fix a single threshold for the sum of the Euclidean differences in the 3 color channels as in:

$$\frac{1}{n^2}[||P_i(L)-P_j(L)||^2+||P_i(U)-P_j(U)||^2+||P_i(V)-P_j(V)||^2] < T$$

then the similarity in one color channel may compensate for the difference in another, producing skewed results (see fig. 10).

Multiple color channels are possible, but we choose to work in the (CIE)LUV color space, which is known to be more perceptually uniform than the standard RGB color space [**?**]. Additionally, our formulation makes it possible to impose separate distance thresholds on each of the color channels (T_1, T_2, T_3). However, for simplicity of analysis, we set $T_1 = T_2 = T_3 = T$, where the choice for the value of T is described next.

6.4 Distance Threshold

Choosing a threshold T requires weighing the quantitative benefits of compression with the qualitatively poorer image reconstructions. We ran a number of experiments, varying the threshold, and quantitatively and qualitatively examining the results. We rescale each color channel to be between 0 and 1, in order to choose a threshold T that is bounded by these values. In fig. 11 we plot a few small experiments (with 200 images) for demonstrative purposes. The images were 500×500 pixels, and the patch size was 25×25. We chose this patch size due to the discussion in 6.1. Below we consider a number of quantitative indicators for patch compression.

In the set of graphs in the first column of fig. 11, we plot in blue the dictionary size against the number of images added to the database. We compare this to the total number of patches that would have been stored if no compression scheme was utilized (red dotted line). We can see that for all choices of threshold, the size of the patch dictionary grows slower than the total number of patches that would need to be added if all images were stored along with their original patches. The gap between the blue and red dotted lines is a measure of the storage savings. As the threshold becomes more stringent, the blue line approaches the red dotted line. Note additionally that as the threshold becomes smaller, patches are more likely to be reused from the same image than from other images when compressing an image, because only patches from the same image will be similar enough to other patches from this image.

The second column of fig. 11 contains histograms indicating how many images contributed different amounts of new patches to the patch dictionary. When the threshold is very small (as in the histogram in the last row) more of the images contribute most of their patches (380-400 new patches added to the patch dictionary per image). Note additionally that the small number of images that are contributing 0 new patches to the dictionary account for the 10% duplicates that are present in the SUN database (more about this in sec. 9.1).

The third column of fig. 11 contains an insertion history: for each image inserted into the database, we track how many of its patches were added to the patch dictionary. We

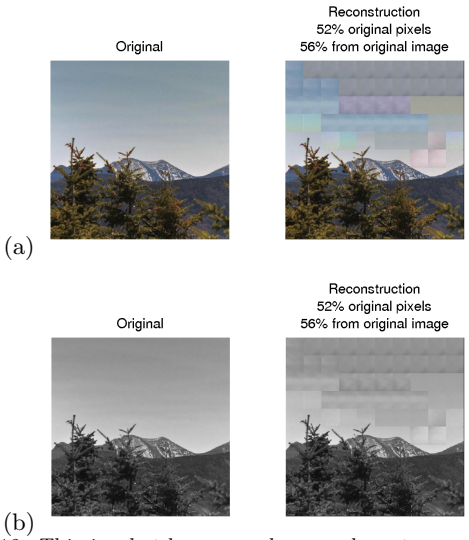


Figure 10: This is what happens when we do not separately constrain each of the color channels to match. We have patches that are (a) the wrong color and produce visible visual artifacts, while (b) matching in terms of general hue (average of the color channels). Again, the patch size and distance threshold were chosen to emphasize the artifacts.

can see that as the patch distance threshold decreases, most images contribute most of their patches. This provides similar information as the histogram (which is merely the cumulative), but allows us to monitor any temporal changes. The spikes to 0 in this graph are indicative of duplicate images, and are discussed further in sec. 9.1.

In the final column of fig. 11, we see a sample image reconstruction. We can see that the reconstruction quality increases, and visual artifacts decrease, as the distance threshold decreases (becomes more stringent). At some point in the middle, the reconstruction is already indistinguishable from the original, but with significant database compression benefits. Thus, for further analysis we consider the threshold $T = 0.01$ (recall that this is per color channel, and is independent of patch size).

6.5 Generalization across image categories and sizes

In the preceding sections, we determined that for an image sized 500×500, a 25×25 patch size with a $T = 0.01$ patch distance threshold is appropriate since compression savings are properly balanced against artifacts introduced during image reconstruction. For further experiments, we scale down the image size to 100×100 and the patch size to 5×5, accordingly, maintaining the patch:image size ratio. Note that the distance threshold still applies as it is independent of patch size. The reduction in image size allows us to consider larger experiments on databases of image thumbnails.

In fig. 12 we consider the growth of the patch dictionary as we add 2K images to our database. We add images from different categories, and observe a slight change in dictionary growth rate for every new category added. Although there is some variation across categories, the overall compression is 18.95% per image, demonstrating generalization across categories.

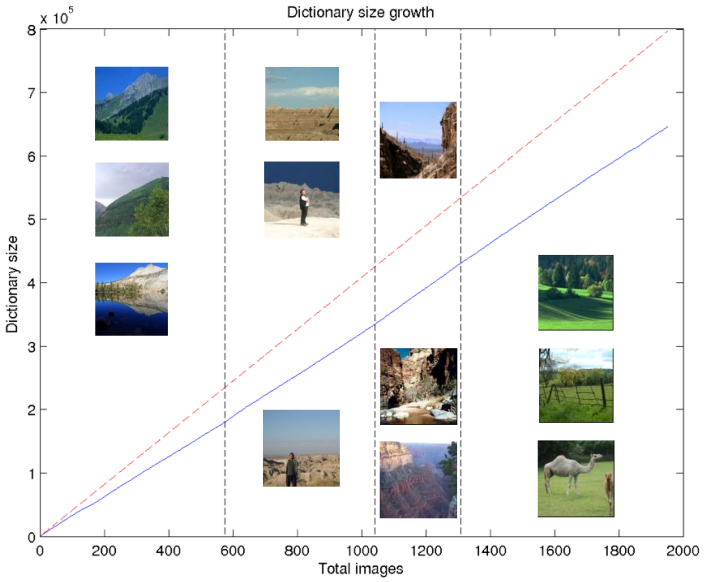


Figure 12: A patch dictionary constructed from 5×5 patches from 100×100 images, with a $T = 0.01$ patch distance threshold, obtains an average compression of 18.95% per image. The average number of new patches added per image is 330.70 with a total dictionary size of $645,534$ patches representing 1952 images. The black dotted vertical lines mark the addition of new image categories, (left to right) mountain (with dictionary growth rate 312.96, equivalent to compression of 21.76% per image), badlands (growth rate 330.50, compression 17.38%), canyon (growth rate: 360.61, compression 9.85%), and pasture (growth rate: 334.11, 16.47%). A few image samples from the 4 categories are included. Note that for this growth rate, storing image patches will always take less space than storing the entire image, since $c < c'$.

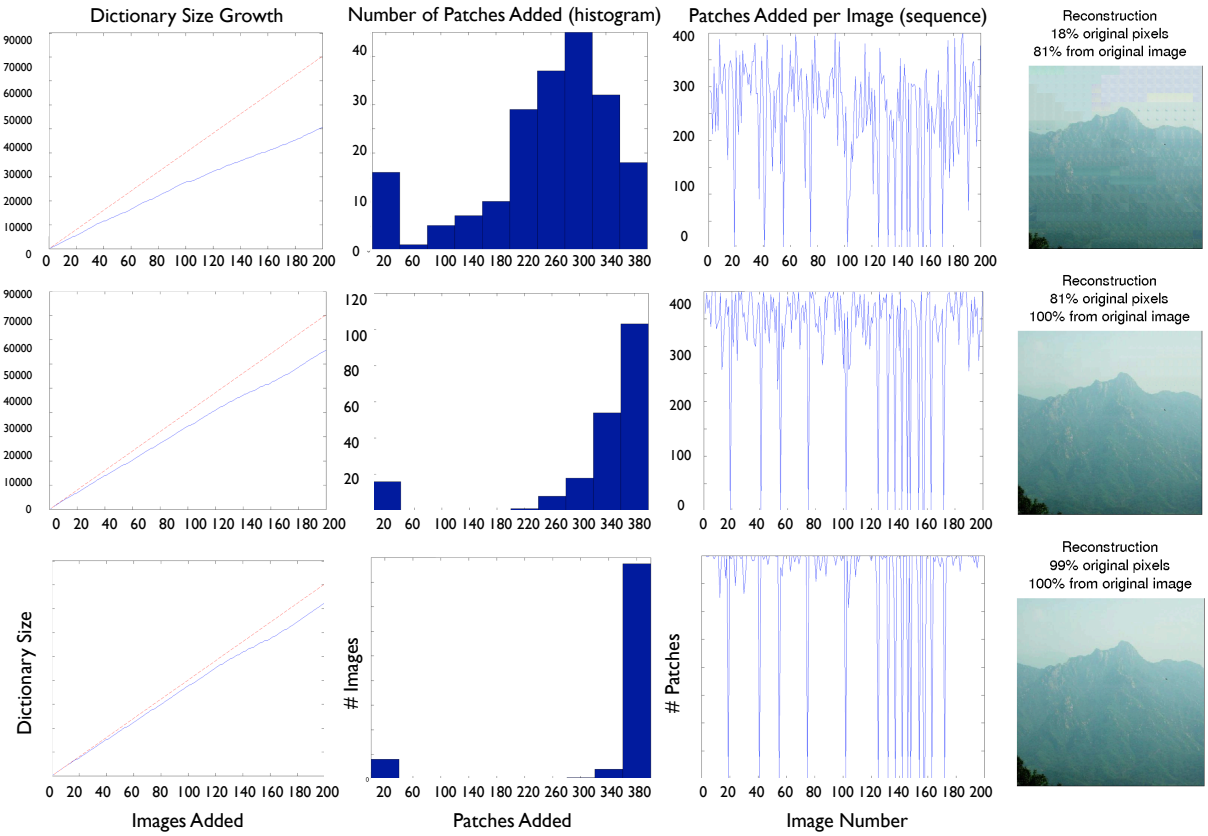


Figure 11: Quantitative and qualitative results obtained by varying the patch distance threshold, T, while extracting 400 patches from each image. In the first row, $T = 0.1$, the dictionary size is 50,822 patches, and the average compression per image is 36.73%. In the second row, $T = 0.01$, the dictionary size is 65,794 patches, and the average compression per image is 18.09%. In the third row, $T = 0.001$, the dictionary size is 72,445 patches, and the average compression per image is 9.80%.

Not all image content is amenable to the same type of compression. For some types of images, particularly where there is a lot of spatial structure (e.g. indoors scenes with objects and parts), compression artifacts are much more noticeable and thus distance thresholds should be more stringent. An interesting extension that is beyond the score of this paper would be to have a content-aware distance threshold (self-adjusting to content type such as indoor vs outdoor/natural).

For our database system, compression works for both indoor and outdoor images. In outdoor images, a lot of the compression savings come from accounting for the homogeneous sky patches (which may take up a large portion of the image). Although indoor images often contain many more objects and are more cluttered than outdoor images, they contain large homogenous regions corresponding to the walls and ceilings of rooms, which can surprisingly compress better than some outdoor images (walls may be more homogeneous than the sky). Thus, our compression approach generalizes to different types of scenes.

7. QUANTITATIVE EVALUATION

Once the quality threshold is chosen, the key differentiator between the speed of database construction and the quality of reconstructed images is the hashing strategy. We constructed 3 databases on the same set of 10,000 images sampled from all categories of the SUN database, with image size set to 500, and patch size set to 25, using the following hashing strategies for Near Neighbor(NN) search:

1. **Naive NN**: 10 random projection vectors sampled from unit Normal with uniform bin size (outliers truncated)
2. **PCA NN**: 10 first principal components as projection vectors with bin size adapted to the distribution of projections
3. **PCA + U NN**: nearly uniform patches are hashed using Luv color quantization into 864 bins, and non-uniform patches are handled with PCA NN

We detail a qualitative evaluation of the results with a user study in sec. 8. Here we consider the quantitative performance of our system as we grow our database to 10K images.

In fig. 13 we see that as for the smaller tests in sec. 6, the growth of our patch dictionary is sub-linear, demonstrating increasing compression benefits as more images are added to the database. We show that this trend holds for all 3 of our NN methods (*naive*, *pca*, and *pca+u*). Note that the *naive* NN approach became infeasible as the dictionary grew, as can be demonstrated by the time required to upload each successive image into the database (see fig. 14). Due to this behavior, it was terminated early. We see that the two approaches using the *pca* NN scheme have significantly better time performance. This is because in the *naive* approach, most of the data falls in relatively few bins. Making use of *pca* helps by projecting data onto the directions of maximal variance, which helps to differentiate between patches, and thus redistribute them across multiple bins. Table 1 includes a breakdown of the number of patches and bins produced in each of the NN approaches, as well as the timings to insert images. Figure 15 demonstrates that in the *naive* approach, the percent of patches that fall into the first bin already

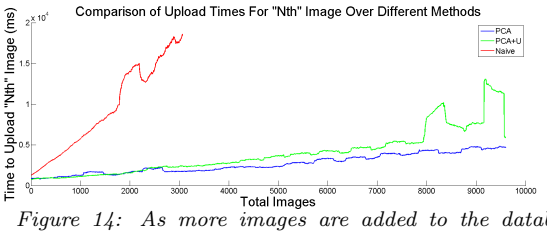


Figure 14: As more images are added to the database, more patches are available in the dictionary. Thus, it is expected that we will need to look through more of the patches in order to determine if a matching patch already exists or if a new one should be added. The time to insert becomes infeasible in the case of the naive *approach because many patches end up in the first few bins (see fig. 15) requiring a linear search through all of them. The pca-based approaches allow patches to be more evenly distributed across bins which reduces the look-up time during insertion (fewer patches to examine in any given bin). Note: the weird behavior at the end of the green curve is an artifact of another process coming online at the same time as the database insertion procedure was run.*

ranking	requirement
5	no visible distortions
4	only minor distortions
3	distortions present, but objects are recognizable
2	pretty large distortions, but scene is recognizable
1	severe distortions, scene is not recognizable

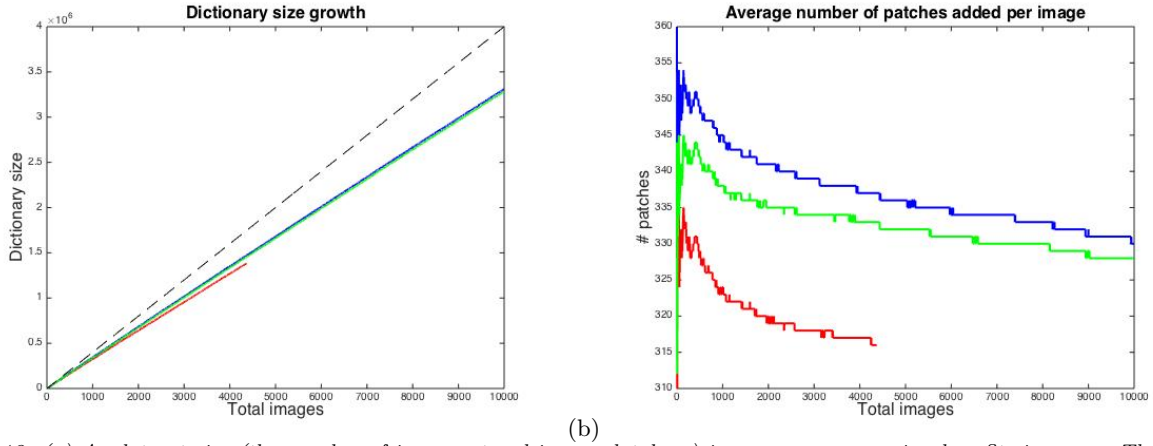
Table 2: The quality evaluation scheme used.

exceeds 50%, and over 70% of the dictionary's patches are accounted for by the third bin. This explains why the *naive* approach takes such a long time to run: it must go through a very big number of patches to determine whether to add new patches from an inserted image. Because the *naive* approach has fewer bins, it is also easier to find a matching patch in a bin, and thus new patches are less frequently added to the dictionary, leading to higher compression ratios using this approach. This explains the gap we see between the red and green/blue curves in fig. 13a. Another way to look at this is to consider the average number of patches added per image (see fig. 13b). As more and more images are added to the database, fewer patches are added to the dictionary. We can more clearly see the gap between the red and green/blue curves in this plot. Even though we do not see the dictionary size plateauing even for our 10K image database, the trends in fig. 13b are promising, and show a decline in the number of patches being added. We believe that much bigger datasets should be investigated in future work to really see the plateau effect and gain the full benefits of our patch-based compression system. Note, however, that even personal image collections are often bigger than 10K images, so this is not an unreasonable requirement.

8. QUALITATIVE EVALUATION

For the qualitative evaluation, 3 of the paper's authors independently rated 100 randomly-selected image reconstructions from each of the three nearest neighbor methods described in sec. 4: *naive*, *pca*, and *pca+u* (all 3 raters received the same set of images). Tabletab:rankings outlines the rating scheme used.

None of the raters received any instructions other than the rating scheme, and the evaluation was not further discussed. Nevertheless, all the raters were highly correlated with each



(a) (b)

Figure 13: (a) As dataset size (the number of images stored in our database) increases, compression benefits increase. The dictionary size is likely to plateau for even larger databases. (b) As dataset size increases, the average number of patches added for each new image decreases. This demonstrates that we are effectively making use of patch redundancy across images. In both plots, the red line corresponds to the naive *NN approach, the blue to the* pca *NN approach, and the green to the* pca+u *NN approach. We can see that the naive approach obtains better compression, but at a high timing cost, as depicted in fig. 14 and explained further in fig. 15. The pca-based approaches allow our system to be feasible.*

	Up to 4.4K			Up to 10K		
method	**time**	**#patches**	**#bins**	**time**	**#patches**	**#bins**
naive	18h20min	1,384,080	670,928	n/a	n/a	n/a
pca	1h54min	1,473,592	1,282,768	7h27min	3,309,583	2,751,235
pca+u	2h5min	1,456,059	1,275,827	10h56min	3,281,241	2,742,888

Table 1: Results on 10,000 images samples from all the categories of the SUN database, where the rows are for naive projection hashing, PCA-based hashing and PCA-based hashing combine with uniform patch hashing.

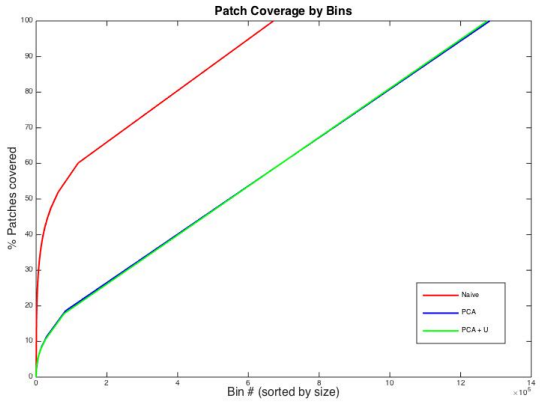


Figure 15: Most of the patches hash to the first few bins in naive *NN approach, which explains why this approach takes a very long time to insert new images into the database. The pca-based methods end up with fewer patches per bin, which is crucial to making our whole pipeline feasible.*

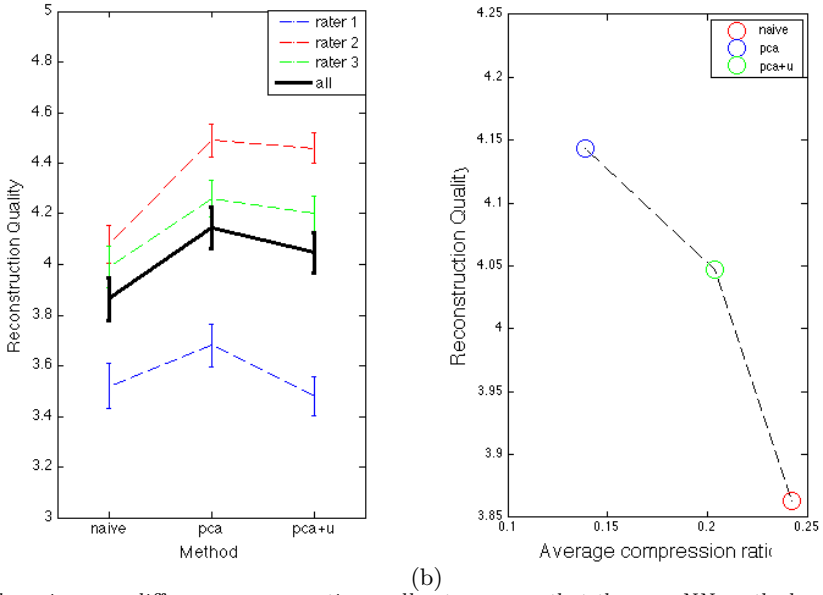
other, where the inter-rater correlation ranged from 0.71 to 0.75 on the *naive* images, 0.51 to 0.61 on the *pca* images, and 0.59 to 0.64 on the *pca+u* images. In figure 17a we plot the mean and standard error image rating for each rater and each NN method. We see some differences between the raters, but most notably, the *pca* approach was rated highest across all 3 raters, and when taking the average ratings (over all 3 raters), the reconstruction quality is statistically significantly better rated for the *pca* method over the *naive* method.

Next, to determine whether ratings corresponded to the actual compression ratio for the images, we correlated each of the rater's scores with the image compression ratios, and then considered different combinations of the rater scores (min, max, mean, mode, median) to see which statistic over the subjective ratings might be most predictive of the objective image compression (see table 3). Because raters are highly subjective as to what they consider a reasonable reconstruction (even given the criteria above), we will use the mean rating across all raters as it is most correlated with image compression, and most consistent. In figure 16 we demonstrate the *pca* NN method reconstructions of a set of images with large compression ratios, that also received high quality ratings (taking the mean across raters). This figure gives a little more insight about how our system works, because naturally, the images with the highest ratings are going to be the ones that benefit least from compression, and that is a less interesting case to consider. We are interested in reasonable reconstructions with great cost savings. We plot the trade-off between compression ratio and reconstruction quality in fig. 17b. As expected, quality ratings go down as the compression ratio increases. Depending on

method	rater 1	rater 2	rater 3	min	max	mean	mode	median
naive	0.7337	0.6700	0.6203	0.7247	0.6837	**0.7484**	0.6324	0.6636
pca	0.7450	0.7916	0.5061	0.6999	0.6791	**0.8070**	0.6695	0.7349
pca+u	0.6169	0.6685	0.6934	0.6340	0.6755	**0.7635**	0.6043	0.7276

Table 3: Here we determine what function of rater scores is most predictive of image compression ratios. We see that the mean across the rater scores has the highest correlation with image compression, presumably because it filters out the noise inherent in subjective judgements.

Figure 16: Here we plot 6 sample reconstructions (using the pca *NN method) that had high compression ratios* and *relatively high ratings (mean over 3 raters).*



(a) (b)

Figure 17: (a) Although there is some difference across ratings, all raters agree that the pca *NN method produces the highest quality reconstructions. (b) Reconstruction quality is inversely proportional to compression ratio. Different applications might desire different settings.*

the application, the optimal point can be chosen.

9. CONCLUSION

We have presented in this paper a largely scalable lossy compression scheme which exploits indexing infrastructure provided by database management systems for efficient storage, and exploits the redundancy of large image databases to drastically reduce the storage cost. In doing so, we compared three methods which exploited locality sensitive hashing for image indexing - a naive method which relied on random projections, a method which exploited the dimensionality reduction of principal component analysis, and a second PCA-based method which also aimed to make the storage process more efficient and provide qualitative image quality improvements by recognizing uniform patches in LUV color-space. Our experiments demonstrated that the cost of image insertion for our PCA-based hashing grows at a much slower rate than the naive method. Meanwhile, qualitative evaluation demonstrated that the reconstruction quality improved with our advanced methods while sacrificing little in additional patch storage. Our methods allowed for patch matching far faster than brute force methods, which early experiments proved to be infeasibly slow.

A few open questions remain. Although most insertion strategies are likely bounded by a quadratic asymptotic runtime, it remains an open question what the *expected* runtime of our hashing based insertion methods - although we empirically see the cost of each additional image scaling roughly linearly in database size, the trend remains to be proven. While our insertion method is much faster than the brute force method, which we found to be intractably slow for our purposes, it is an open question if a method exists which can still provide good compression and database growth guarantees while having a constant insertion cost.

Finally, although we saw storage improvements in a regime of 10,000 images with good compression quality, we anticipate these trends to be more pronounced for very large databases of 10,000,000 images or more. More experiments still need to be run to validate this belief empirically.

9.1 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 and similar image retrieval.

9.1.1 Duplicate Detection

Encoding images as pointers to a collection of patches provides the ability to quickly spot images that contain large overlapping regions (composed of the same patches). In the extreme case, if multiple images point to the same set of patches, then we know these images are duplicates. Duplicates are a big problem in big computer vision datasets because they occur frequently and are hard to manually remove. They occur frequently (sometimes up to 10% of the time) because these datasets are automatically scraped from the internet, where the same image can occur under separate identifiers (on different websites, copied and uploaded by different users, etc.). The SUN database [**?**] used in this paper is no exception.

Duplicates are difficult to detect because not all duplicates are pixel-wise identical: the same image encoded using different standards or sized to different dimensions (even when resized to the same dimension later) will look almost identical to the human eye, but will contain different pixel values. Our patch distance metric is forgiving to perturbation at the pixel-level as long as the patch is overall similar to another patch (see sec.6.3). If multiple images map to the same set of patches that means that the corresponding patches in those images are within a distance threshold of each other (upper-bounded by $2T$). If multiple images map to all of the same patches, then we have good guarantees that the images are near-duplicates. Otherwise, the probability that every single patch matched would be low (i.e. low that two images are similar locally, for multiple local locations - as many locations as patches).

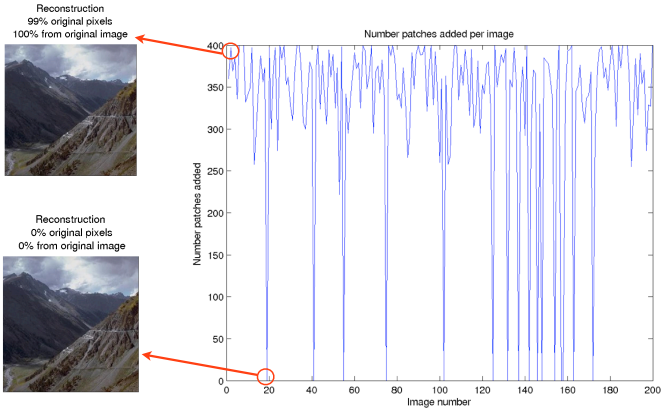


Figure 18: This is an example of the first 200 consecutive insertion queries to an empty database: for each image inserted, we can measure how many new patches were added to the patch dictionary (out of 400 patches in the image). When we see that this number spikes down to 0 we know that the image has been fully reconstructed from patches from other images. We can check if all those patches came from a single other image. If that is the case, we know we have a duplicate or near-duplicate image.

We can use these properties to spot duplicates in our database on-the-fly. For instance, when an image is added to the database, we can measure how many new patches the image contributed to the patch dictionary (because similar-enough patches could not be found), and how much of the image was mapped to pre-existing dictionary patches. When an image is reconstructed fully from the dictionary patches, and the patches it is reconstructed from all come from a single other image in the database, we know that the newly-added image is a duplicate. This is depicted in fig. 18.

By the same logic, similar images are those that overlap in terms of the patches they share in common. We can easily compare the two patch pointer vectors of two images to check their overlap. We can check if this overlap corresponds to patches clustered together in the images (for instance, when only some local region of the images matches, like when they share an object). We can thus discover images that have different degrees of overlap with other images.

9.1.2 Photomosaics

One interesting (and somewhat whimsical) application of our system is in the automated fabrication of photomosaics

from images. A photomosaic [2] is an image which is created by partitioning a pre-existing 2-D piece of artwork into small, equally sized rectangles. Each of these rectangles is then replaced with a small image which approximates the original color and texture of the rectangle, keeping the overall artwork recognizable. Thus, the final result is an image composed of hundreds of smaller images.

Our system can be directly applied to the synthesis of such images, with a few very small modifications. First, instead of determining which patches to store based on our previous dictionary and image collection, we a priori store a selected input image set as our patch dictionary, but scaled to "patch size." For demonstration purposes, we created our patch dictionary by scaling down and storing the entire SUN database. In order to create the photomosaic, we first choose and store a target image we would like to transform. In storing the image, we, as usual, split the image into patches, and for each patch perform a nearest neighbor search. However, in this case, if no patch already exists in the hashed bin, we expand our nearest neighbor search to more bins until we find a bin with at least one patch, and choose the most similar one. During this step, we *never* store patches; we always map the patch to one already in the dictionary. The mapping output by this process results in a photomosaic.

We demonstrate this process on a 1600x1200 image of the Stata Center at MIT, using 25x25 rectangular patches. We show the original image and reconstruction in figure 19.

9.2 Future Extensions

In this paper we considered a simple implementation of a patch-based image database compression scheme, where the images and patches were square and of fixed sizes. Patches were sampled in a regular, non-overlapping grid from each image. Alternative approaches include more flexible, context-aware, patch-sampling techniques. For instance, the patch granularity for sampling large homogenous sky and field regions may be different from the one used for sampling highly-textured regions like objects and structures (trees, buildings, people, etc.). Similarly, patches that do not cross object boundaries are likely to lead to less artifacts in future reconstructions. For this, approaches like Selective Search [**?**] that localize image regions likely to contain objects, may prove promising for sampling patches.

If patches were different sizes, then one of a number of extensions to the system would be required - for instance, (1) a patch transformation scheme, or (2) a hierarchical patch dictionary. A patch transformation scheme would permit each patch to be transformed (in a simple way - e.g. via rescaling) to match another patch with the same appearance but different (scale) parameters. For instance, a small patch in one image may be sufficient to account for a much larger part of another image, and rather than storing many separate patches of different sizes, we would benefit from quickly applying transformations to existing dictionary patches. To implement this system would require storing, for each image location, not only a pointer to a patch in the patch dictionary but also a transformation (e.g. a set of scaling parameters). Naturally, the cost function (to weigh the benefits of such a scheme vs storing the original images or even equally-sized patches) would need to take into account (a) the extra parameters stored along with each image location, and (b) the reconstruction time overhead for patch transformation. With large enough datasets, this approach may be effective at eliminating redundancy.

Another approach, building hierarchical patch dictionaries, may speed up the patchifying and subsequent reconstruction of an image, by offering a top-down approach. If larger patches match, there is no need to parse the image at a finer-grained scale. Only if large patches do not properly account for the structure in an image, would it be necessary to go to a finer-grained patch size. Note that small patches could be composed into larger patches, via a hierarchy, so that if large-patch matches are not found, descending down the patch hierarchy of the best-matching large patches would make it possible to find a match at a lower granularity. This scheme would be less flexible than the patch transformation scheme, but may prove to be more efficient.

[2] See, for example, `http://en.wikipedia.org/wiki/Photographic_mosaic`.

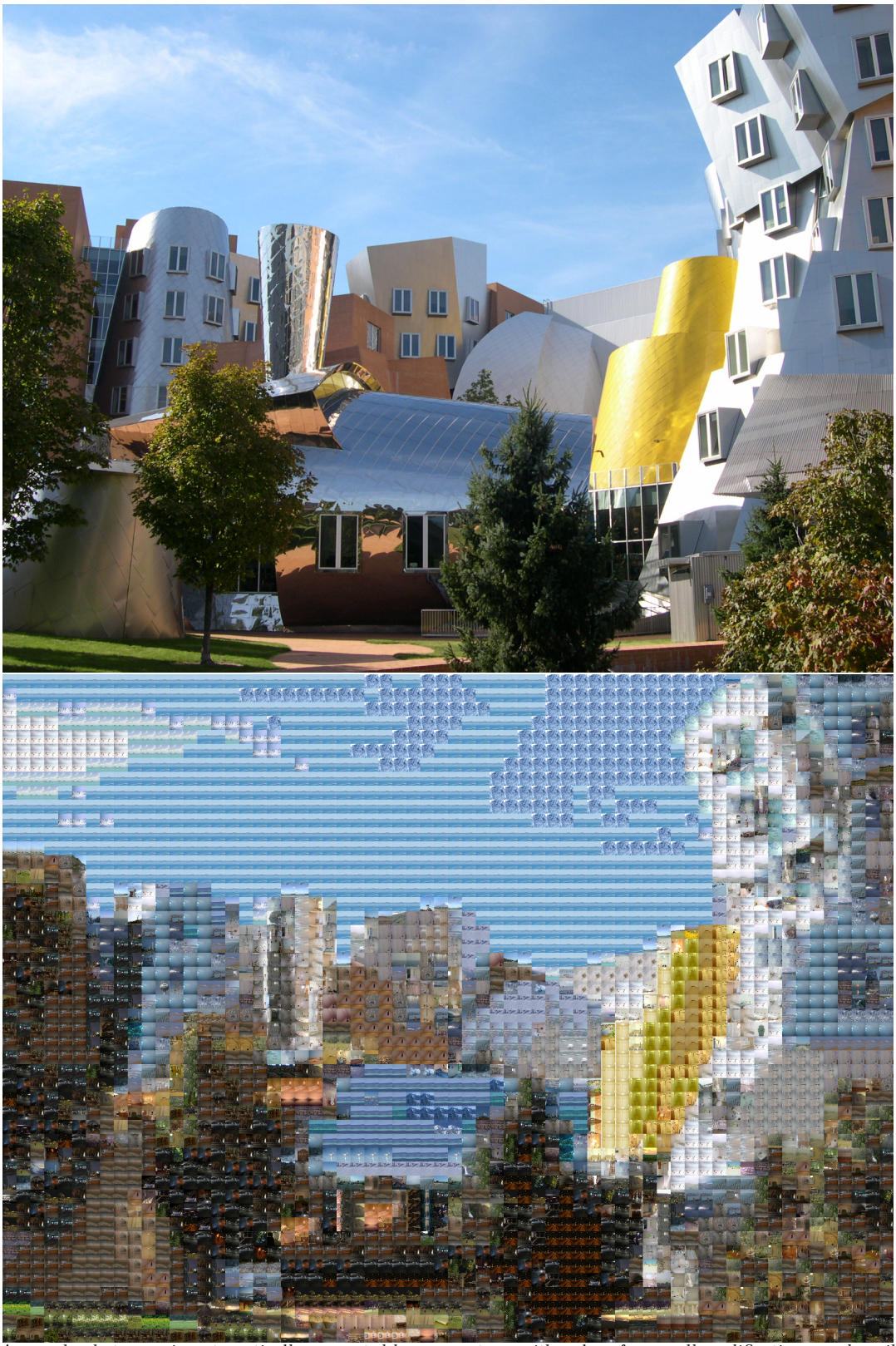


Figure 19: A sample photomosaic automatically generated by our system, with only a few small modifications as described in 9.1.2.