Building and Using a Database of One Trillion Natural-Image Patches

Sean Arietta and Jason Lawrence ▪ *University of Virginia*

Many example-based image-processing algorithms operate on image patches (texture synthesis, resolution enhancement, image denoising, and so on). However, inaccessibility to a large, varied collection of image patches has hindered widespread adoption of these methods. The authors describe the construction of a database of one trillion image patches and demonstrate its research utility.

Research over the past 10 years has demonstrated the utility of example-based techniques in solving image-processing problems. These have proven useful in areas such as texture synthesis,[1] resolution enhancement,[2] image denoising,[3] and hole filling.[4] These problems are mathematically ill-posed, in that the desired output contains more information than the input. So, any algorithm looking to solve them must incorporate assumptions about our world to select a plausible result from the set of algebraic solutions. For example, to produce useful output for resolution enhancement, the underlying algorithm must "understand" what different objects look like over a range of spatial scales.

Approaches that use a collection of training images to construct either a *regularization term* in an objective function, a *constraint* in a search, or a *prior* image model in a Bayesian setup, often outperform traditional tools based on "analytic" image models.[5] However, these approaches require access to a large, diverse set of training images and efficient search tools. This search often occurs at the patch level, decomposing an input image into a set of partially overlapping patches and searching a database of training images for similar patches. Recent work shows that in order to achieve reliable inference, this database should grow exponentially as the patch size increases.[6] However, owing to the immense storage and computational costs of processing large repositories of high-resolution images, researchers typically evaluate their work using only tens to hundreds of training images—a small fraction of what's required.

Fortunately, the growing availability of large distributed (cloud) computing resources allows access to more training data than ever before. Companies such as Google, Amazon, Yahoo, Microsoft, and IBM maintain large data centers the public can access through programs such as Amazon's Elastic Compute Cloud (http://aws.amazon.com/ec2), Windows Azure (www.microsoft.com/azure), and the US National Science Foundation's Cluster Exploratory program (CluE; www.nsf.gov/clue).

Exploiting this opportunity, we built a database of one trillion (10^{12}) natural-image patches and a search system that performs nearest-neighbor (NN) queries. We evaluated our search algorithm's performance on two popular parallel computing architectures. Our database has proven useful for studying the fundamental relationships between the patch size, amount of training data, and nearest neighbors' expected quality. We've proposed the first analytic expression relating these three quantities. This expression lets us predict any one of the quantities from the other two to provide a baseline measurement of the performance of any patch-based image-processing algorithm.

The Database

We constructed our database from one million images downloaded from the Internet. Here we

 Published by the IEEE Computer Society

look at the hardware and software architectures we used and at the database layout.

Distributed-Computing Architectures

We used two separate clusters, each representing a different hardware architecture and programming framework. IBM and Google provided the first cluster through the CluE program. The Hadoop 0.20.0 programming framework and the Hadoop Distributed File System (HDFS) manage access to this cluster, and the code is a combination of Java and C.

The Hadoop framework exposes the classic map/reduce functional approach to distributed computing and mimics Google's MapReduce programming framework. In such setups, the data to process resides on the local disks of individual machines, and the infrastructure copies executable code to whatever machine contains the portion of the database it intends to process. This differs from more traditional grid systems that use a dedicated high-bandwidth network to copy data from a central repository to the compute nodes' main memory. The HDFS has built-in redundancy and fault tolerance. Each physical machine contains seven Intel Xeon 64-bit processors running at 2.8 GHz with 8 Gbytes of shared memory. Altogether, this cluster consists of 416 compute nodes that can execute 834 map operations in parallel and 830 reduce operations in parallel.

We also implemented our database on the Ranger cluster at the Texas Advanced Computing Center (TACC; www.tacc.utexas.edu), part of the TeraGrid infrastructure (www.teragrid.org). Ranger has 3,936 compute nodes. Each node is a SunBlade x6420 with four AMD Opteron Quad-Core 64-bit processors (16 cores total) running at 2.3 GHz with 32 Gbytes of shared memory. These nodes are connected through an InfiniBand switch with 1-Gbyte-per-second unidirectional point-to-point bandwidth. Altogether, Ranger has 62,976 processing cores, 123 Tbytes of distributed memory, and 1.7 Pbytes of shared disk storage managed with the Lustre parallel file system. MPI manages access to these machines, and the code is in C.

Database Layout

We're interested in image-processing systems that construct their output patch-by-patch. Such systems have three key parameters: the patch size, the information (feature vectors) and distance function used to identify similar patches, and the available training images' size and diversity.

To study these parameters, we favored a design flexible enough to allow modifying any of them at search time. So, we store a set of complete images rather than precomputing patches and the corresponding feature vectors. If these parameters were fixed, precomputing and storing this information along with any search acceleration data structures would significantly decrease running time.

We organized the image set in a straightforward hierarchical layout. We store each image as a JPEG file, using its original compression parameters. Each set of 1,000 images forms a leaf folder. Each set of 10 leaf folders resides in a parent folder, each set of 10 parent folders resides in another folder, and so on, up to a root folder containing all the images. Unlike a flat layout, this scheme avoids limiting the number of files in a directory and allows more efficient traversal of a subset of the database.

Our database has proven useful for studying the fundamental relationships between the patch size, amount of training data, and nearest neighbors' expected quality.

We created our database by downloading the images from Flickr (www.flickr.com). To maximize bandwidth, we downloaded them in parallel to a distributed file system according to our hierarchical layout. Downloading 1 million images took approximately five hours on the Hadoop cluster and approximately three hours on Ranger. We excluded any image larger than 4,096 pixels in either dimension but applied no other filters. We also restricted our download to images published under the Creative Commons licensing rules (http://creativecommons.org).

The average image in our database is roughly 1,000 pixels in each dimension and thus requires approximately 1 Mbyte of storage (so, all one million images occupy approximately 1 Tbyte). Each image contains roughly one million individual patches, if you assume a distinct patch at each pixel (in other words, the patches fully overlap), providing approximately 10^{12} patches. To the best of our knowledge, this is roughly four orders of magnitude larger than datasets used in previous patch-based image-processing systems.[2,3,7] (For more information on related research, see the sidebar.)

Searching the Database

The central computational task we face is identifying the set of patches in our database that are similar to those in a query image. Fortunately,

Related Work Involving Large Image Collections and Image Patches

Recent projects show compelling applications of large image collections. James Hays and Alexei Efros described a system that completes missing image regions by searching for similar entries in a database of one million images.[1] Researchers have also provided novel 3D interfaces for browsing image collections[2] and have created new images by using parts of other images.[1–7] Such systems would benefit from efficient access to a large database of image patches.

Our empirical study on the relationship between patch size and the amount of training data required to achieve accurate matches is related to research studying the statistics of natural image patches. Early research examined the second-order statistics using methods such as singular value decomposition and independent component analysis. Bruno Olshausen and David Field estimated an overcomplete basis that produces a sparse representation of image patches conjectured to resemble the early stages of human vision.[8] More recently, Damon Chandler and Field estimated the entropy (information content) of 3 × 3 and 8 × 8 grayscale image patches.[9] Our experiments extend this research for larger color image patches, providing the basis for an analytic function relating the fundamental relationship between patch size, amount of training data, and the expected accuracy of nearest neighbors.

References

1. J. Hays and A.A. Efros, "Scene Completion Using Millions of Photographs," *ACM Trans. Graphics*, vol. 26, no. 3, 2007, article 4.
2. N. Snavely, S.M. Seitz, and R. Szeliski, "Photo Tourism: Exploring Photo Collections in 3D," *ACM Trans. Graphics*, vol. 25, no. 3, 2006, pp. 835–846.
3. A.A. Efros and T.K. Leung, "Texture Synthesis by Non-parametric Sampling," *IEEE Int'l Conf. Computer Vision*, IEEE CS Press, 1999, pp. 1033–1038.
4. W. Freeman, E. Pasztor, and O. Carmichael, "Learning Low-Level Vision," *Int'l J. Computer Vision*, vol. 40, no. 1, 2000, pp. 25–47.
5. A. Buades, B. Coll, and J.M. Morel, "A Review of Image Denoising Algorithms, with a New One," *Multiscale Modeling and Simulation*, vol. 4, no. 2, 2005, pp. 490–530.
6. M. Elad and D. Datsenko, "Example-Based Regularization Deployed to Super-resolution Reconstruction of a Single Image," *Computer J.*, vol. 50, no. 4, 2007, pp. 1–16.
7. J. Yang et al., "Image Super-resolution as Sparse Representation of Raw Image Patches," *Proc. 2008 IEEE Conf. Computer Vision and Pattern Recognition* (CVPR 08), IEEE CS Press, 2008, pp. 1–8.
8. B.A. Olshausen and D.J. Field, "Sparse Coding with an Overcomplete Basis Set: A Strategy Employed by V1?" *Vision Research*, vol. 37, no. 23, 1997, pp. 3311–3325.
9. D.M. Chandler and D.J. Field, "Estimates of the Information Content and Dimensionality of Natural Scenes from Proximity Distributions," *J. Optical Soc. America A*, vol. 24, no. 4, 2007, pp. 922–941.

such NN searches arise in many fields and have undergone extensive study. NN algorithms are broadly classified according to whether they return an exact or approximate solution. Example-based image-processing tasks almost always require a set of representative matches rather than an exact set. So, we investigated approximation algorithms and avoided exact NN algorithms, whose runtime costs grow prohibitive as the dimensionality of the search space (patch size) increases.

We focus on the k-NN problem, which involves computing the k patches in our training database closest to a query patch, as determined by the chosen feature vector and similarity function. (This differs from the ϵ-nearest neighbor problem, which computes the set of patches that are within distance ϵ from the query.)

A feature vector describes the information in one patch and is often a key design decision in these systems. An example of a simple feature vector is the concatenation of the patch's raw intensity or RGB pixel values. We compute feature vectors by removing low-frequency information from each patch. This separation is common in many missing-data problems because high-frequency information is often corrupted or missing. We isolate these frequency bands by subtracting an image from a version of itself that has been convolved with a wide Gaussian filter.[2] Two patches are similar if and only if the L_2 distance between their corresponding feature vectors is small. The running times we report would be the same for any feature vector of the same size because our design allows modifying the feature vector without additional processing.

We evaluated two state-of-the-art algorithms for approximate k-NN searches. One algorithm uses a kd-tree to partition the space around a set of points in conjunction with a priority queue to accelerate searches.[8] (We use the terms "point," "feature vector," and "image patch" interchangeably because you can consider a patch's feature vector to be a point in $\mathbb{R}^d$.) The other algorithm, which we ultimately chose for our system, is based on *locality-sensitive hashing* (LSH).[9] For search spaces of very large dimension, such as the ones we consider, LSH is theoretically superior to methods that partition space.[9]

Locality-Sensitive Hashing

Consider a feature vector $\hat{x} \in \mathbb{R}^d$ (for example, in the case of 4 × 4 RGB patches, $d = 48$). LSH computes the hash function:

$$h(\hat{x}) = \left\lfloor \frac{\hat{x} \cdot \hat{a} + b}{w} \right\rfloor, \quad (1)$$

where $\hat{a}$ is a vector in $\mathbb{R}^d$ constructed by sampling a p-stable distribution (for $p = 2$, the L_2 norm, this is a normal distribution with zero mean and unit standard deviation[9]), w is a scalar-valued parameter determining the size of the interval along $\hat{a}$ that gets hashed to the same integer, and b is a uniform random variable in the range $[0, w]$.

LSH indexes a hash table using a key formed by combining K values of this function for different $\hat{a}$ and b. We follow the method that Alexandr Andoni and Piotr Indyk proposed, which computes L separate hash tables.[9] Given a query point $\hat{q} \in \mathbb{R}^d$, LSH computes a hash key for each of the L tables, performs a brute-force search of the contents of the buckets into which the key falls, and returns the k (or fewer) nearest neighbors encountered during this search.

Figure 1 illustrates each parameter's role in the LSH algorithm: w, K, and L. The parameter w controls the maximum distance a neighbor can be to the query, because they can't hash to the same integer (see Equation 1).

Increasing K increases the chance that the final brute-force search won't consider distant irrelevant points. Geometrically, projecting a point onto K lines is akin to approximating the ball of radius w centered at the query by the visual hull formed by intersecting orthographic views from these lines (see Figure 1). This approximation's accuracy improves with more lines. So, K controls a trade-off between the cost of indexing each L hash table and the final search's efficiency (measured as the number of neighbors found relative to the number of points compared).

Increasing L decreases the chance that the search will miss a point within distance w from the query simply because that point happens to straddle the boundary between two consecutive intervals along one of the lines $\hat{a}$. Larger values of L produce more accurate searches but longer running times.We found it useful to terminate the brute-force search early to guarantee that processing time for one query image didn't exceed a limit. This is another mechanism for approximating the search, because the search will return only the k nearest neighbors encountered up to the limit. This modification helped achieve good load balancing.

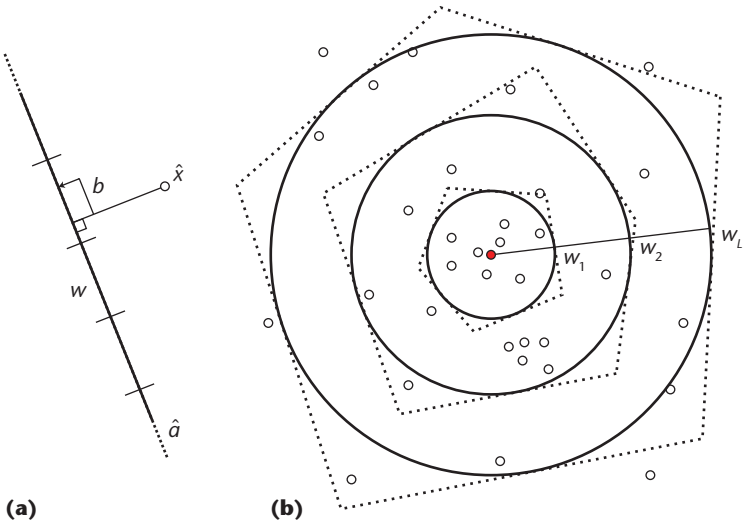


Figure 1. The locality-sensitive hashing (LSH) algorithm computes the set of points (black circles) near a query (the red circle). (a) The algorithm projects each point onto a set of *K* lines. (b) This process basically approximates a ball of radius *w* around the query (the solid line) by its visual hull (the dashed line). The set of points that project to the same segments as the query are searched for neighbors. This process repeats *L* times, using potentially different values of *w* as shown here.

Researchers have proposed sophisticated strategies for computing K and L, given a fixed w.[9] These strategies minimize the probability that the solution will exclude a nearest neighbor while maintaining near-optimal time and space efficiency. We opted instead for a simple trial-and-error strategy, and determined that $K = 20$ and $L = 4$ worked well for the searches we wished to perform. However, we had difficulty finding a value of w that worked well across different patch sizes, feature vectors, and images. Zhou Wang and his colleagues also noted this difficulty.[10] Following their recommendation, we use a different value of w for each hash table, such that $w_1 < w_2 < \ldots < w_L$. This approach, combined with early termination, adapts the search according to the density of patches in the area around a given query patch. Again, through trial-and-error, we found that a schedule of $w_1 = 1.0$, $w_2 = 5.0$, $w_3 = 10.0$, and $w_4 = 40.0$ worked well.

Performance Analysis

We compared methods for computing neighbors that use

- the standard LSH algorithm,
- a version of LSH that uses different values of w for each hash table, and
- a popular implementation of the kd-tree-based algorithm distributed with the ANN (approximate nearest neighbor) library.[8]

Figure 2. The four images used in our experiments. We compiled a set of test images that exhibited variation in the amount of high-frequency content they contained. The town image, for example, contains a significant amount of high-frequency information while the fish image contains mostly low-frequency information.

Using $k = 100$, we measured the running time and accuracy of computing the neighbors of 1,216 query patches (randomly sampled from a test image) in a 512 × 512 training image (roughly 300,000 training patches). We recorded our results for four test images (see Figure 2) and 100 training images, which we randomly selected from our database and downsampled to 512 × 512. Although this amount of data is less than we typically process, the kd-tree algorithm's memory requirements proved infeasible for larger test sets.

We measured accuracy in two ways. *Match accuracy* is the percentage of true nearest neighbors returned. *Region difference* is the percentage increase in the distance between the query and the most distant neighbor in the set of k matches, as compared to an exact search. So, a perfect search would have a match accuracy of 100 percent and a region difference of 0 percent. We found that region difference better indicates the results' visual quality because it indicates how dissimilar returned matches are to an exact search. Using match accuracy alone to evaluate different methods can be misleading because it tends to underestimate performance.

Algorithms adjust the degree of approximation in different ways. Users of the kd-tree algorithm can specify an *approximation factor* ϵ. This terminates the search as soon as the ith neighbor found is guaranteed not to exceed the true distance to the real ith nearest neighbor by a factor of $(1 + \epsilon)$. Using $\epsilon = 0.0$ results in an exact search. LSH's accuracy, on the other hand, is inherently approximate for any values of L and K. We held these values fixed and controlled the degree of approximation by adjusting early termination deadlines.

Figure 3 presents our experiments' results for 6 × 6 patches ($d = 108$) and 12 × 12 patches ($d = 432$). Figure 3a plots the match accuracy as a function of the total running time (the time required to construct the acceleration data structure and traverse it). Figure 3b plots the region difference as a function of the running time. We chose values for the approximation factor and early termination deadline to cover a similar range of running times. As we expected, the kd-tree algorithm achieves an exact match with the longest running time. The two LSH algorithms never achieve perfect accuracy, although they come close. Table 1 reports the algorithms' memory use.

What Figure 3 doesn't show is the breakdown of the total running time into construction and traversal times. For the 6 × 6 patches, building the kd-tree took 11.3 seconds and building the hash tables took 1.4 seconds. For the 12 × 12 patches, these times were 26.3 and 2.8 seconds, respectively. These experiments confirm that the kd-tree algorithm is more appropriate when fine-control over accuracy is important and that LSH is more appropriate when space constraints exist. They also confirm that LSH grows more accurate as the degree of approximation and patch sizes increase. In addition, our adaptive LSH offers a modest improvement over standard LSH for very short running times. The results we report in the rest of this article refer to adaptive LSH.

Distributed Processing

Despite LSH's efficiency, searching all the patches in a million 1-megapixel images would be impractical on a single machine. However, distributing this task across multiple machines (or processes) working in parallel is straightforward.

We compared the distributed-processing performance of the Hadoop and Ranger clusters. In these experiments, a single "job" consisted of computing the approximate set of k nearest neighbors for each patch in a query image. A job executed in three stages. The setup phase located and distributed executables and data across the cluster. The map phase executed many small programs in parallel to locate nearest neighbors in individual database images. The reduce phase aggregated the matches output from the map phase into a set of results, which were written back to the distributed file system.

Most of the computation occurred in the map phase, in which a series of tasks ran in parallel. Each map phase consisted of these tasks:

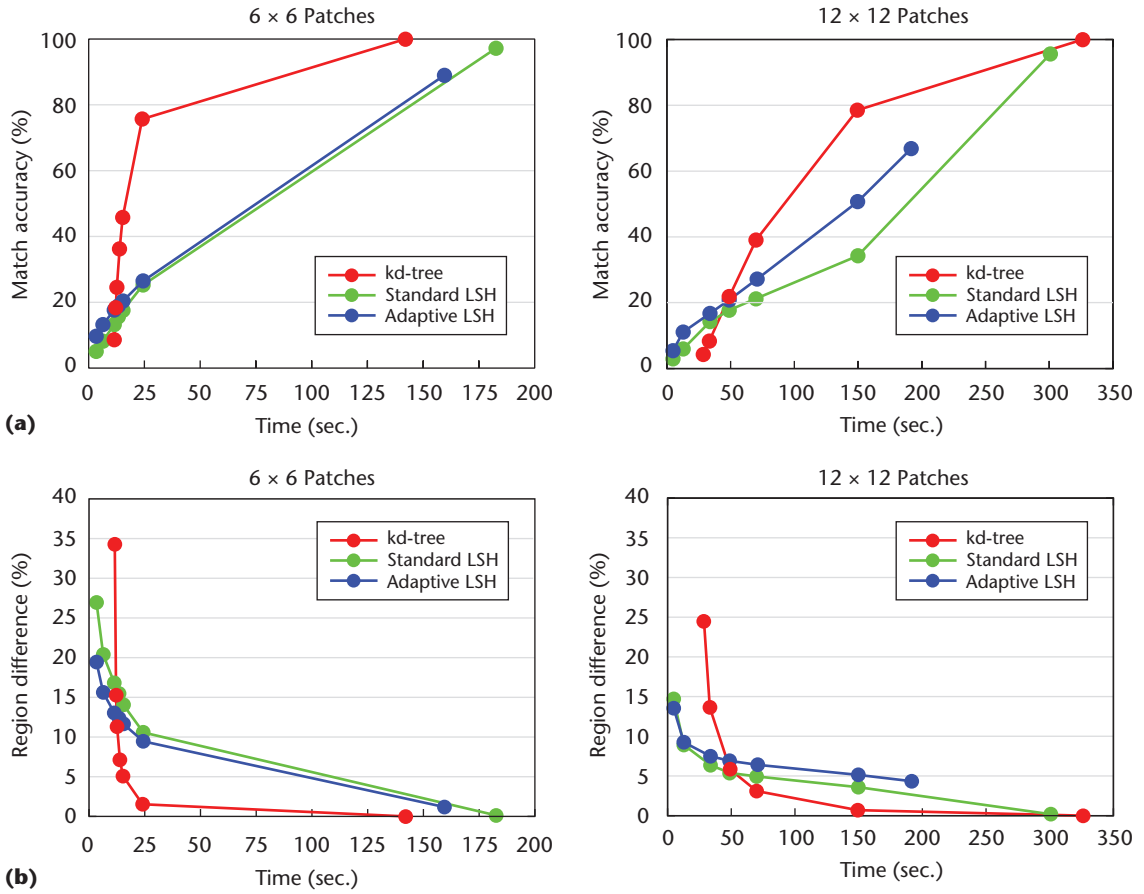


Figure 3. The performance of three nearest-neighbor algorithms. (a) Match accuracy as a function of running time. (b) Region difference as a function of running time. The kd-tree algorithm achieves an exact match with the longest running time. The LSH algorithms never achieve perfect accuracy but come close.

1. Load a JPEG image from the distributed file system and decompress it into memory using the routines in Java's `ImageIO` library (Hadoop) or the `libjpeg` library (Ranger).
2. Construct feature vectors for each patch in the image, which involves performing a series of convolutions using the `libfftw3` library.
3. Build L hash tables and initialize them using the feature vectors from the previous step.
4. Compute each patch's nearest neighbors, using the hash table. This might terminate early to meet deadlines for processing single training images.
5. For Hadoop, send matches to the reduce layer for assembly by the underlying infrastructure. For Ranger, update a fixed-length results table to include matches that are closer than those previously recorded.

We implemented operations 2 through 5 in C. To provide a fair comparison, we used the Java Native Interface with the Hadoop implementation to execute the same code as the MPI implementation in the Ranger cluster. Because the Hadoop cluster couldn't process images larger than 1,024 × 1,024 in the available heap space, we downsampled each image in both clusters to this size before running our experiments.

Table 1. The Memory Use of Three Nearest-Neighbor Algorithms.

	Patch size	
Algorithm	**6 × 6**	**12 × 12**
kd-tree	305 Mbytes	1.1 Gbytes
Locality-Sensitive Hashing	69 Mbytes	70 Mbytes

Recall that the reduce phase merges the map phase's results into a single consolidated results table. In Hadoop, this requires copying the map task outputs to a single reduce task, sorting them according to computed patch distances, and reducing them to a single set of results. To provide a fair comparison, we performed the same operations in our MPI/C implementation, although the sorting step wasn't necessary.

We compared these two implementations' performance on databases of 1,000 images and 10,000 images. The same image served as a query in each case. Figures 4, 5, and 6 report the results for the setup, map, and reduce phases. Because the map phase executes in parallel, each job's total running time depends on the number of processing units

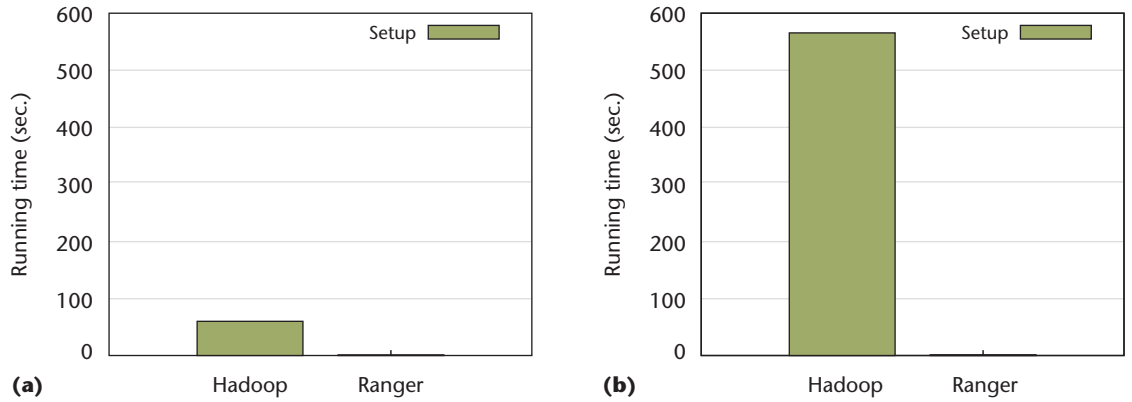


Figure 4. Performance during the setup phase for the Hadoop and Ranger clusters, for (a) 1,000 and (b) 10,000 images. The Hadoop setup time scales linearly with the input size; Ranger setup times are negligible in comparison. Hadoop incurs larger processing times associated with locating and dispatching jobs to the input on the Hadoop Distributed File System.

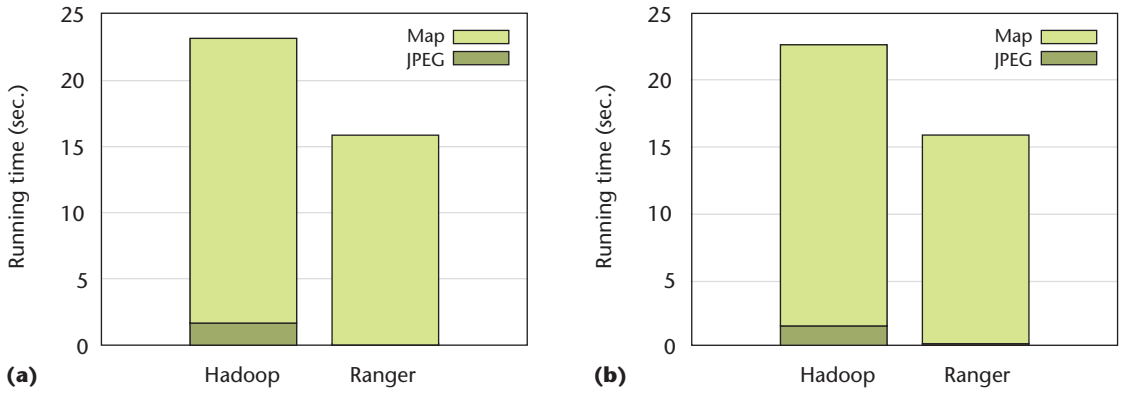


Figure 5 Performance during the map phase for the Hadoop and Ranger clusters, for (a) 1,000 and (b) 10,000 images. "JPEG" is the time spent reading and decoding a JPEG image. Because our nearest-neighbor search terminates after 10 seconds, this plot measures Hadoop's and Ranger's relative efficiency in distributing this computation over multiple processing units. In both clusters, this computation scales effectively with the input size.

allocated to the job. The total (wall clock) time for the entire job is

$$T_S + T_R + \frac{N * |T_M|}{M},$$

where T_S is the setup time, T_R is the reduce time, $|T_M|$ is each map task's average running time, N is the number of training images, and M is the number of available processing units.

We used one process for each image on both the Hadoop cluster and Ranger cluster. On the Ranger cluster, we requested 32 processing cores for the 1,000-image database and 128 processing cores for the 10,000-image database. Since Hadoop has no way to guarantee an allotment of processing cores, the framework is responsible for distributing the images across the available nodes. The elapsed wall clock times for Ranger to execute one search over the two databases were approximately 9 minutes and approximately 22 minutes. The corresponding times for Hadoop were approximately 14 minutes and approximately 43 minutes (however, these times are less indicative of performance compared to those found in Figures 4, 5, and 6 for the reasons stated above.)

We also measured each map task's computational throughput as the number of distance calculations per second. Recall that map tasks terminate after a fixed deadline that's the same in both setups. So, larger computational throughput will produce higher-quality results because a task will consider more patches. Table 2 reports these rates, which indicate that, on average, our MPI/C implementation performs roughly twice as many distance calculations as our Hadoop implementation.

On the basis of these results, we conclude that although both implementations perform comparably in the map phase, the setup and reduce phases are considerably faster on MPI/C than on Hadoop/Java.

In our algorithm's final version, we terminated

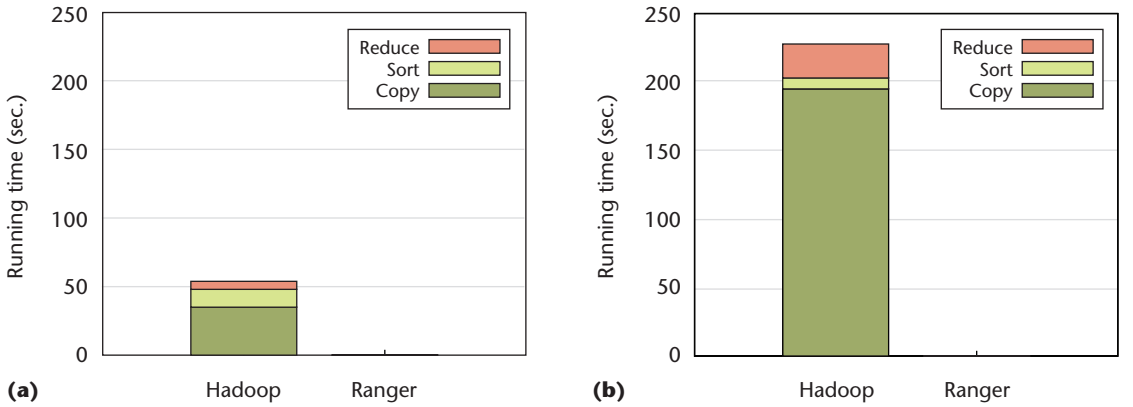


Figure 6. Performance during the reduce phase for the Hadoop and Ranger clusters, for (a) 1,000 and (b) 10,000 images. This phase consists of a copy, a sort, and a reduce operation. The copy operation on Hadoop is significantly longer than on Ranger. This is due to a combination of lag in the setup and map phases (the copy operation must wait for these to complete) and the narrower bandwidth of the interconnect in the Hadoop cluster.

the search for neighbors in each training image after 10 seconds. This deadline resulted in an average match accuracy of 12 percent and an average region difference of 15 percent (these numbers vary with patch size; see Figure 3). These 10 seconds are in addition to the time spent decoding the image and building the hash tables, which weren't limited in any way. Altogether, processing a training image took 20 seconds on average. Because the search time is bounded, this total time depends very weakly on the size of the query image and of the patch. In all, we successfully processed one million training images in under two hours using 4,096 processing cores.

Patch Size and Match Accuracy

The reconstruction granularity, or patch size, is a fundamental parameter in any example-based image-processing system. A patch size that's too small will adversely affect the system's ability to capture long-range structures in the image. When the patch size is too large, the number of examples needed to find high quality matches is infeasible.

In a series of experiments, we reconstructed a set of patches from a query image with patches from a set of training images. More precisely, we replaced high-frequency information in each query patch with corresponding high-frequency information in the nearest patch from our database. We isolated images' high-frequency information using the common frequency separation procedure we described previously. Figure 7 shows our interpretation of these low-frequency and high-frequency image subbands for one of the images.

This reconstruction's accuracy visually indicates how densely the space of natural-image patches is sampled in our database. Poor reconstructions indicate a large average distance between nearest neighbors and thus an insufficient sampling.

Table 2. The Average Number of Distance Computations.

	Database size	
Framework	**1,000 images**	**10,000 images**
Hadoop	1.64×10^7	1.65×10^7
Message-passing interface	2.13×10^7	2.22×10^7

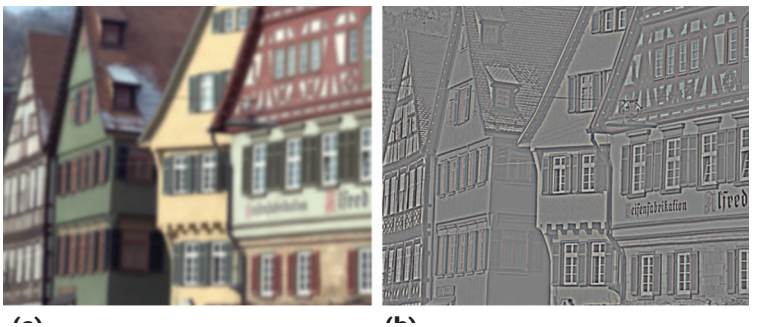


Figure 7. Subband decomposition. We constructed a query from the town image by separating the (a) low-frequency and (b) high-frequency subbands. The sum of the low-frequency and high-frequency images reproduces the original image. We reconstructed the high-frequency data in a query image using nearest neighbors from the database for different patch sizes.

Conversely, good reconstructions indicate a small distance between nearest neighbors and thus an adequate sampling.

It's worth noting that we aren't trying to directly reconstruct images from other images. Solving that problem is trivial using 1 × 1 patches (single pixels) and a relatively small image database. Instead, our experiments aim to derive an analytic expression that allows predicting one of the parameters (the patch size, training-data size, or average nearest-neighbor distance) from the other two.

We used the four query images in Figure 2 and patches of sizes 4 × 4, 6 × 6, 8 × 8, and 10 × 10 that overlapped by a single column or row of pixels on all sides. For each patch in a query image, we

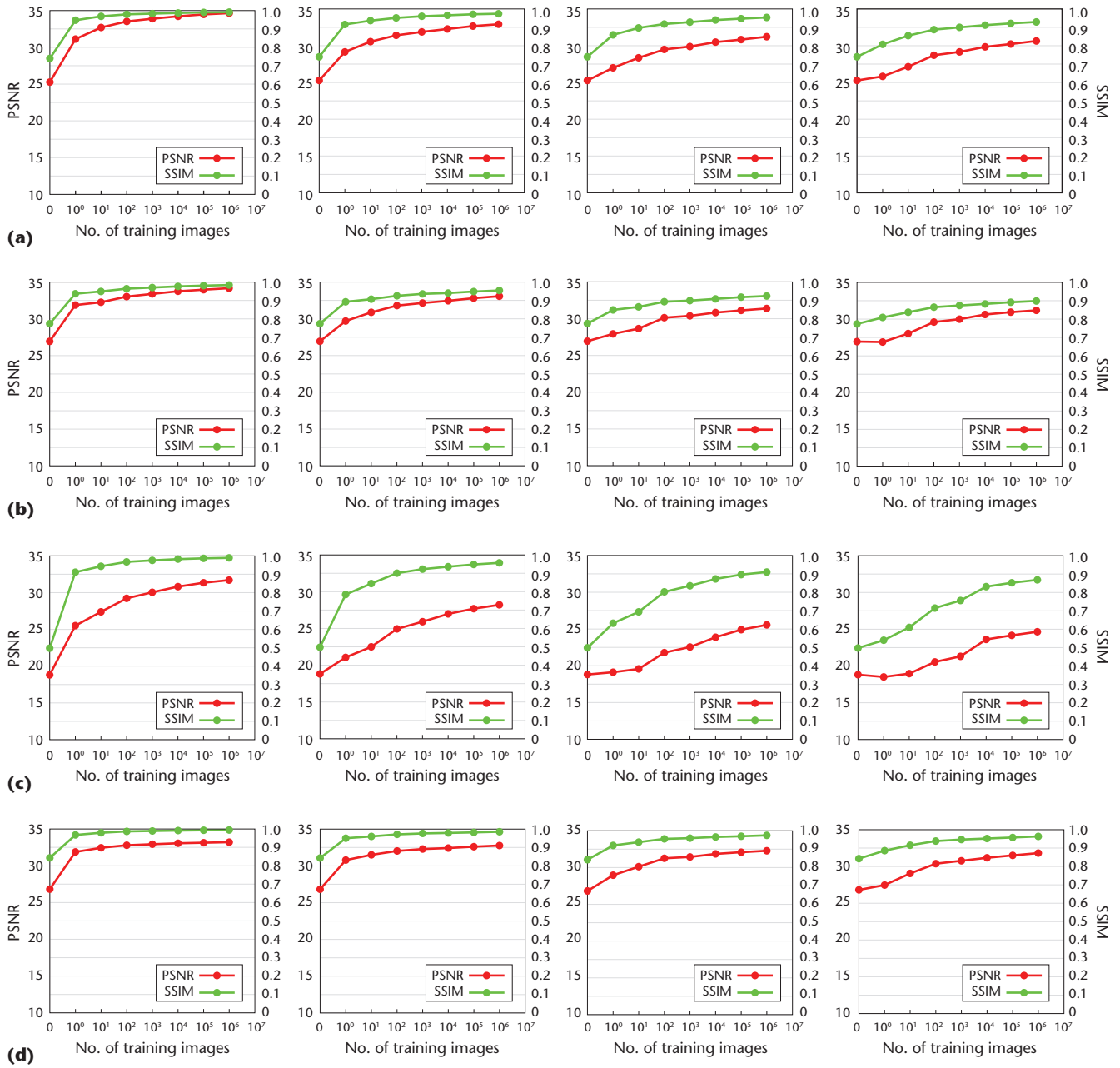


Figure 8. The image quality of reconstructed high-frequency subbands for different patch sizes and amounts of training data, for the (a) flower, (b) girl, (c) town, and (d) fish images. PSNR and SSIM stand for peak signal-to-noise ratio and structural similarity index. The graphs show that reconstruction accuracy increases as the number of training images increases, but at a decreasing rate.

computed its single nearest neighbor from either one, ten, one hundred, one thousand, ten thousand, one hundred thousand, or one million training images randomly chosen from our database. We then combined these nearest neighbors into a complete image by averaging the values along the boundaries where patches overlapped.

We retained low-frequency information from the query and reconstructed only high-frequency bands. This reconstruction corresponds to the best result achievable with a particular training set and patch size. In other words, the difference between the original query image I and the reconstruction $\tilde{I}$ is an upper bound on the performance of any example-based system that uses raw image patches directly. Of course, there's no guarantee that an algorithm can achieve this optimal reconstruction when data is missing from the query (as is the case with image denoising, superresolution, and colorization) because you'll never know with certainty which image patch you're looking for. However, this reconstruction reveals the experimental conditions required for optimal reconstruction to be possible.

The graphs in Figure 8 plot the difference be-

tween I and $\tilde{I}$ in the form of the peak signal-to-noise ratio (PSNR) and the structural similarity (SSIM) index.[10] These graphs show a clear trend: reconstruction accuracy increases as the number of training images increases, but at a decreasing rate. Also, certain images require more data to reconstruct than others for the same patch size and amount of training data. We attribute this to the fact that some images (such as the town image) contain significantly more high-frequency information than others consisting of smoother low-frequency gradients (for example, the fish image). Figure 9 also confirms the unsurprising fact that reconstructing images using larger patches requires more training data.

To extrapolate these results to larger training sets and patch sizes, we experimented with analytic expressions that link the expected SSIM, denoted by q, to the number of patches in the training set n. We had the best success with a simple rational function:

$$q = 1 - \frac{1}{a_d \log n + b_d},$$

where the dependence on d (the dimension of the patches) is made explicit in the parameters a_d and b_d. We fit these parameters to the average SSIM (computed over the four test images) for each patch size using the Nelder-Mead simplex algorithm.

Figure 10 shows the obtained fits, superimposed over the measured averages. Figure 11 shows another view of this data. It plots the amount of training data (measured in image patches) our analytic model predicted would be necessary to achieve different quality scores for different patch sizes. We expect that researchers will use this model to either estimate a lower bound on the amount of training data needed to achieve a given quality score or estimate the largest patch an available amount of training data can reliably support.

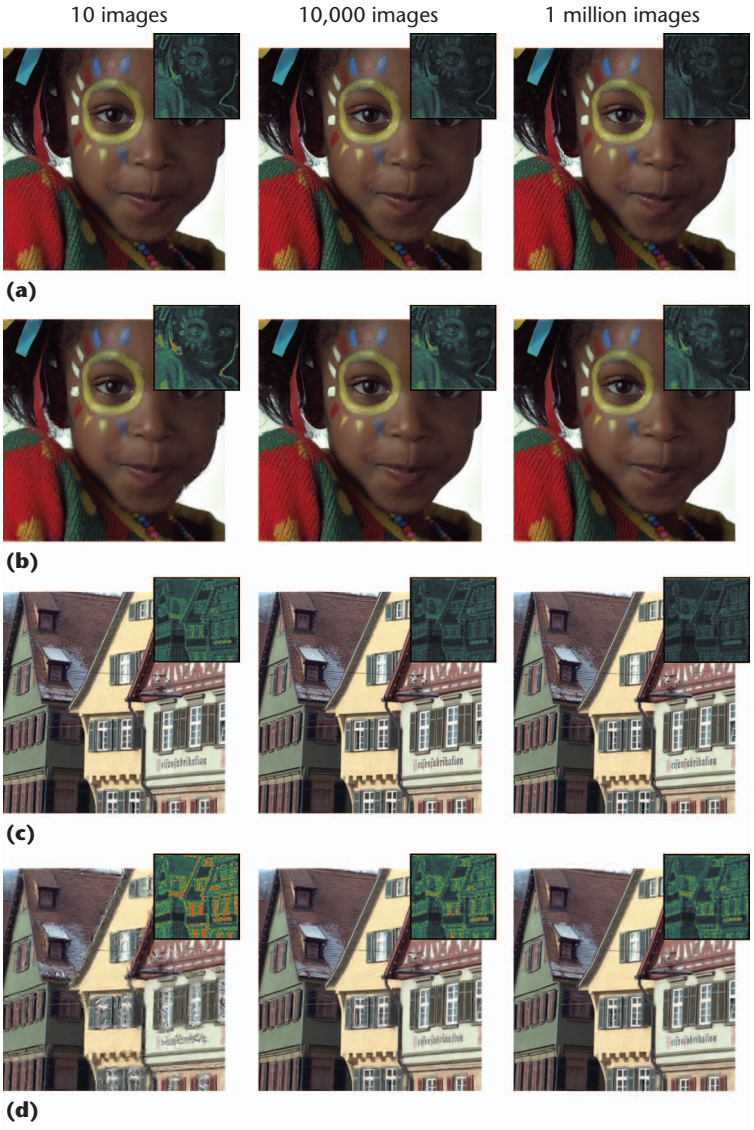
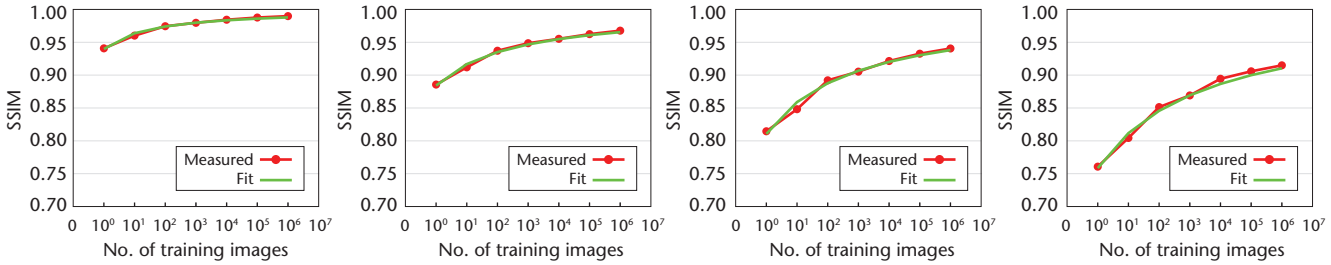


Figure 9. Image reconstructions with difference visualizations, for the girl image using (a) 4 × 4 and (b) 8 × 8 patches and for the town image using (c) 4 × 4 and (d) 8 × 8 patches. These images illustrate the error introduced in reconstructing the girl and town images using these patch sizes across multiple dataset sizes. The results indicate that larger patch sizes require increasingly more data to produce high-quality reconstructions.



(a) ($a = 4.764$, $b = -49.205$) **(b)** ($a = 1.469$, $b = -11.655$) **(c)** ($a = 0.788$, $b = -5.609$) **(d)** ($a = 0.508$, $b = -2.884$)

Figure 10. The expected SSIM score as a function of the number of training images for (a) 4 × 4, (b) 6 × 6, (c) 8 × 8, and (d) 10 × 10 patches. Each graph compares measurements (averages over the four test images) to predictions by our analytic model, $q = 1 - (1/(a \log n + b))$. The best-fitting parameters, which we computed using the Nelder-Mead optimization algorithm, appear below each graph.

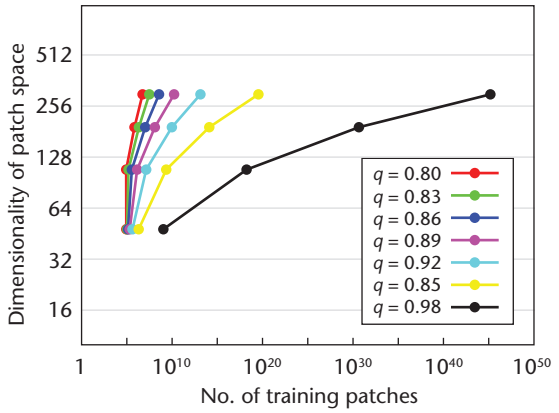


Figure 11. The amount of training data (in patches) necessary to achieve different quality thresholds for different patch sizes. The patch space's dimensionality is equal to the product of the patch width, patch height, and number of color bands.

Our database and search system provide a blueprint for others interested in building similar tools to study other problems in Internet-scale image processing. We also believe our analytic expression relating the patch size, amount of training data, and average degree of similarity between nearest patches will be useful to researchers developing similar systems. We're currently extending our research to investigate specific missing-data problems, including resolution enhancement and denoising.

References

1. A.A. Efros and T.K. Leung, "Texture Synthesis by Non-parametric Sampling," *IEEE Int'l Conf. Computer Vision*, IEEE CS Press, 1999, pp. 1033–1038.
2. W. Freeman, E. Pasztor, and O. Carmichael, "Learning Low-Level Vision," *Int'l J. Computer Vision*, vol. 40, no. 1, 2000, pp. 25–47.
3. A. Buades, B. Coll, and J.M. Morel, "A Review of Image Denoising Algorithms, with a New One," *Multiscale Modeling and Simulation*, vol. 4, no. 2, 2005, pp. 490–530.
4. J. Hays and A.A. Efros, "Scene Completion Using Millions of Photographs," *ACM Trans. Graphics*, vol. 26, no. 3, 2007, article 4.
5. M. Elad and D. Datsenko, "Example-Based Regularization Deployed to Super-resolution Reconstruction of a Single Image," *Computer J.*, vol. 50, no. 4, 2007, pp. 1–16.
6. D.M. Chandler and D.J. Field, "Estimates of the Information Content and Dimensionality of Natural Scenes from Proximity Distributions," *J. Optical Soc. America A*, vol. 24, no. 4, 2007, pp. 922–941.
7. J. Yang et al., "Image Super-resolution as Sparse Representation of Raw Image Patches," *Proc. 2008 IEEE Conf. Computer Vision and Pattern Recognition* (CVPR 08), IEEE CS Press, 2008, pp. 1–8.
8. D.M. Mount and S. Arya. "ANN: A Library for Approximate Nearest Neighbor Searching," 2006; www.cs.umd.edu/~mount/ANN.
9. A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," *Comm. ACM*, vol. 51, no. 1, 2008, pp. 117–122.
10. Z. Wang et al., "Image Quality Assessment: From Error Visibility to Structural Similarity," *IEEE Trans. Image Processing*, vol. 13, no. 4, 2004, pp. 600–612.

Sean M. Arietta *is a computer science PhD candidate at the University of Virginia. His research interests include image enhancement and restoration, texture synthesis, large-scale image processing, and content-based image retrieval. Arietta has a BS in physics from the University of Virginia. Contact him at sma2t@cs.virginia.edu.*

Jason Lawrence *is an assistant professor of computer science at the University of Virginia. His research interests include acquisition of geometry and material properties, representations of realistic material appearance, and interactive and global-illumination rendering algorithms. Lawrence has a PhD in computer science from Princeton University. Contact him at jdl@cs.virginia.edu.*