Nearest Neighbor Search in Google Correlate

Dan Vanderkam
Google Inc
76 9th Avenue
New York, New York 10011
USA
danvk@google.com

Robert Schonberger
Google Inc
76 9th Avenue
New York, New York 10011
USA
robsc@google.com

Henry Rowley
Google Inc
1600 Amphitheatre Parkway
Mountain View, California
94043 USA
har@google.com

Sanjiv Kumar
Google Inc
76 9th Avenue
New York, New York 10011
USA
sanjivk@google.com

ABSTRACT

This paper presents the algorithms which power Google Correlate[8], a tool which finds web search terms whose popularity over time best matches a user-provided time series. Correlate was developed to generalize the query-based modeling techniques pioneered by Google Flu Trends and make them available to end users.

Correlate searches across millions of candidate query time series to find the best matches, returning results in less than 200 milliseconds. Its feature set and requirements present unique challenges for Approximate Nearest Neighbor (ANN) search techniques. In this paper, we present Asymmetric Hashing (AH), the technique used by Correlate, and show how it can be adapted to fit the specific needs of the product.

We then develop experiments to test the throughput and recall of Asymmetric Hashing as compared to a brute-force search. For "full" search vectors, we achieve a 10x speedup over brute force search while maintaining 97% recall. For search vectors which contain holdout periods, we achieve a 4x speedup over brute force search, also with 97% recall.

General Terms

Algorithms, Hashing, Correlation

Keywords

Approximate Nearest Neighbor, k Means, Hashing, Asymmetric Hashing, Google Correlate, Pearson Correlation

1. INTRODUCTION

In 2008, Google released Google Flu Trends [3]. Flu Trends uses query data to estimate the relative incidence of Influenza in the United States at any time.

This system works by looking at a time series of the rate of Influenza-Like Illness (ILI) provided by the CDC[1], and comparing against the time series of search queries from users to Google Web Search in the past. The highest correlating queries are then noted, and a model that estimates influenza incidence based on present query data is created.

This estimate is valuable because the most recent traditional estimate of the ILI rate is only available after a two week delay. Indeed, there are many fields where estimating the present[1] is important. Other health issues and indicators in economics, such as unemployment, can also benefit from estimates produced using the techniques developed for Google Flu Trends.

Google Flu Trends relies on a multi-hour batch process to find queries that correlate to the ILI time series. Google Correlate allows users to create their own versions of Flu Trends in real time.

Correlate searches millions of candidate queries in under 200ms in order to find the best matches for a target time series. In this paper, we present the algorithms used to achieve this.

The query time series consists of weekly query fractions. The numerator is the number of times a query containing a particular phrase was issued in the United States in a particular week. The denominator is the total number of queries issued in the United States in that same week. This normalization compensates for overall growth in search traffic since 2003. For N weeks of query data, the query and target time series are N dimensional vectors and retrieval requires a search for nearest neighbors.

A brute force search for the nearest neighbors is typically too expensive (we discuss exactly how expensive below), and so extensive work has gone into developing ANN algorithms which trade accuracy for speed. Our problem is similar to image clustering problems[6], and we examined many of the known tree- and hash-based approaches[5, 4, 7].

Google Correlate presents some unique challenges and opportunities for nearest neighbor search:

1. We compare vectors using Pearson correlation. Because vectors may not contain directly comparable quantities (i.e. normalized query volume and user-supplied data), we prefer to use a distance metric which is invariant under linear transformations. Pearson correlation has this property.

2. We require high recall for highly correlated terms. Recall is the fraction of the optimal N results returned

[1]Available online at `http://www.cdc.gov/flu/weekly/`

.

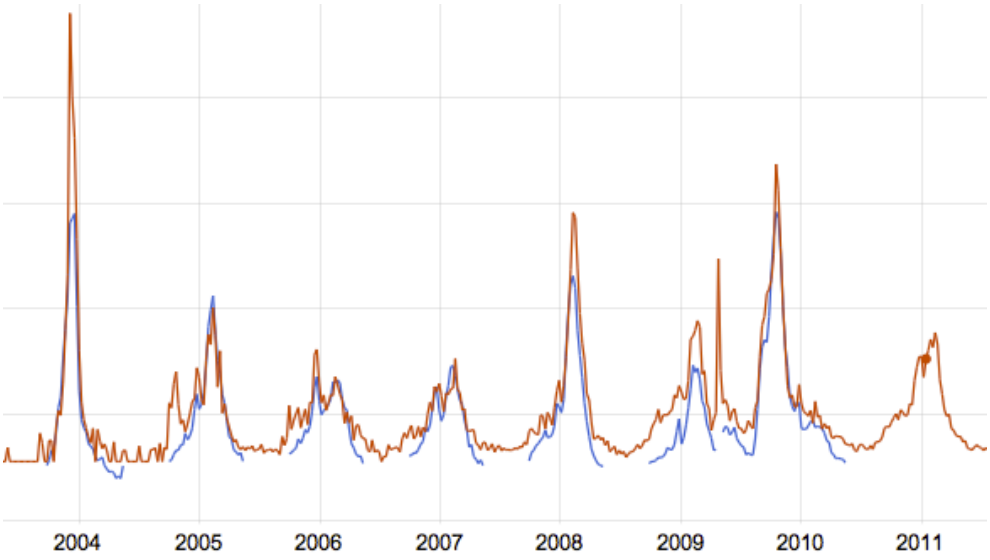

Figure 1: Correlated time series: Influenza-Like Illness (blue) and popularity of the term "treatment for flu" on Google web search (red).

by the system. The original Google Flu Trends model, which served as a template for Google Correlate, consisted of 45 [3] queries. If we only achieved 10% recall, we would have to build a Flu model using only four or five of those queries. This would lead to an unacceptable degradation in quality.

3. We require support for holdout periods. When building a model, one typically splits the time period into a training and test (or holdout) set. The training set is used to build the model and the holdout is used to validate it. Holdouts are essential for validating models, so we needed to support them in Correlate. If a time period is marked as being part of a holdout, our algorithm should not consider any values from that time period.

4. While the data set is large (tens of millions of queries with hundreds of weeks of data each), it is not so large that it cannot be stored in memory when partitioned across hundreds of machines. This is in contrast to a service like Google Image Search, where it would not be reasonable to store all images on the web in memory. Details on how we choose the tens of millions of queries can be found in the Correlate Whitepaper[8].

The solution used by Correlate is a form of vector quantization[2] known as "Asymmetric Hashing", combined with a second pass exact search, which we experimentally find to produce search results quickly and with high recall.

2. DEFINITIONS

2.1 Pearson Correlation

For our distance metrics, we use the standard definition for Pearson correlation between two time series, namely:

$$r(u,v) = \frac{\text{cov}(u,v)}{\sigma_u \sigma_v} \quad (1)$$

$$= \frac{\sum_{i=1}^{n} [(u_i - \mu(u))(v_i - \mu(v))]}{\sqrt{\sum_{i=1}^{n}(u_i - \mu(u))^2}\sqrt{\sum_{i=1}^{n}(v_i - \mu(v))^2}} \quad (2)$$

2.2 Pearson Correlation Distance

If two vectors u and v are perfectly correlated, then $r(u,v) = 1$. A distance function should have a value of zero in this case, so we use the standard definition of Pearson correlation distance:

$$d_p(u,v) = 1 - r(u,v) \quad (3)$$

2.3 Vectors with Missing Indices

Here we formalize the idea of a "holdout" period. We define a vector with missing indices as a pair:

$$(v,m) \text{ where } v \in \Re^N, m \subset \{1 \ldots N\} \quad (4)$$

if an index $i \in m$ then the value of v_i is considered unknown. No algorithm which operates on vectors with missing indices should consider it.

We found this representation convenient to work with, since it allowed us to perform transformations on v. For example, we can convert between sparse and dense representations, without changing the semantics of (v,m).

We also define the projection $\pi(v,m)$ obtained by removing the missing indices from v. If $v \in \Re^N$ then $\pi(v,m) \in \Re^{N-|m|}$.

2.4 Pearson Correlation Distance for Vectors with Missing Indices

Let (u,m) and (v,n) be two vectors with missing indices. Then we define the Pearson correlation distance between them as:

$$d_p((u,m),(v,n)) = d_p(\pi(u, m \cup n), \pi(v, m \cup n)) \quad (5)$$

While this distance function is symmetric and nonnegative, it does not satisfy the triangle inequality. Additionally, if $d_p((u,m),(v,n)) = 0$, there is no guarantee that $u = v$ or $m = n$.

2.5 Recall

We measure the accuracy of approximate result sets using recall, the fraction of the ideal result set which appears in the approximate result set.

If E is the ideal set of the top k results and A is the actual set of the top k approximate results, the we define

$$Recall_k(A,E) = \frac{|A \cap E|}{k} \tag{6}$$

Note that the ordering of results is not considered here. Correlate swaps in exact distances (this procedure is described below), so if results appear in each set, then they will be in the same order.

3. ASYMMETRIC HASHING

Here we present *Asymmetric Hashing*, the technique used by Correlate to compute approximate distance between vectors. We begin by considering the simpler case of Pearson correlation distance with "full" vectors, i.e. those without missing indices. We then adapt the technique for target vectors which do have missing indices.

3.1 Mapping onto Squared Euclidean Distance

We begin by mapping Pearson correlation distance onto a simpler function. If $u, v \in \Re^N$, we can normalize them by their ℓ^2 norms and rescale so that

$$u' = \frac{u - \mu(u)}{2N\,|u - \mu(u)|} \tag{7}$$

$$v' = \frac{v - \mu(v)}{2N\,|v - \mu(v)|} \tag{8}$$

$$\tag{9}$$

then we can calculate that

$$d_p(u,v) = \left|u' - v'\right|^2 \tag{10}$$

Hence we can replace Pearson correlation distance with squared Euclidean distance by normalizing our vectors appropriately. In practice, the database vectors are normalized offline and the target vector is normalized before doing a search. We note here that computing $\mu(u)$ and $|u - \mu(u)|$ makes use of all the values in u. Hence using this normalization is incompatible with holdouts/missing indices.

3.2 Training and indexing

Assume that we have a collection $V = v^{(1)}, v^{(2)}, v^{(3)}, \ldots, v^{(M)}$ of vectors in $\Re^N$. We split these vectors into k-dimensional *chunks* composed of consecutive dimensions in each vector. This results in $\frac{N}{k}$ chunks, as seen in Figure 2.

We can define a set of projections, $\pi_1, \ldots, \pi_{N/k} \in \Re^N \mapsto \Re^k$ which map from the full vectors to each particular chunk. Then, for each $i, \pi_i(V)$ is a set of M vectors in $\Re^k$. There are $\frac{N}{k}$ such sets.

For each of these projections, we run the k-means algorithm to find 256 centroids. We label the j^{th} centroid for the i^{th} projection $c_i^j \in \Re^k$. There are $256\frac{N}{k}$ such centroids. These centroids are the output of the training phase of Asymmetric Hashing.

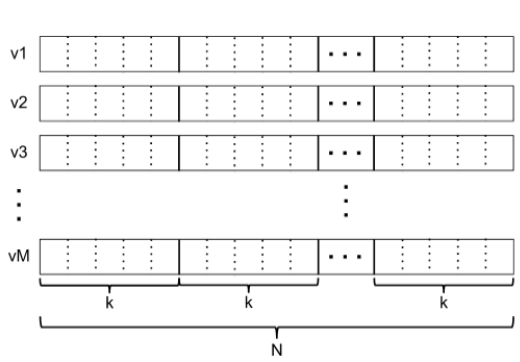

Figure 2: Illustration of how vectors are split into *chunks*

To index a vector v, we find the centroid closest to v in each chunk. Concretely, we set

$$h_i(v) = \arg\min(\left|\pi_i(v) - c_i^j\right|) \tag{11}$$

$$h(v) = (h_1(v), h_2(v), \ldots, h_{N/k}(v)) \tag{12}$$

For each vector this results in $\frac{N}{k}$ integers between 1 and 256, one for each chunk. We combine the integers into a $\frac{N}{k}$ byte hash code, $h(v)$.

We can reconstitute an approximation of the original vector from its hash code as:

$$Approx(v) = (c_1^{h_1(v)}, c_2^{h_2(v)}, \ldots, c_{\frac{N}{k}}^{h_{\frac{N}{k}}(v)}) \tag{13}$$

As an aside, we note that chunks comprised of consecutive weeks (1-10) typically result in better approximations than chunks comprised of evenly-spaced weeks $(1, 41, 81, \ldots, 361)$. This is because real-world time series tend to be auto-correlated: one week is similar to those before and after it. This means that the effective dimensionality of ten consecutive weeks is typically less than ten. The k-means algorithm exploits this to produce accurate approximations.

In this paper, we look at the case where:

$$M = 100000$$
$$N = 400$$
$$k = 10$$

This means that each 400 dimensional vector is indexed to a 40 byte hash code. If the original vectors were stored using 4 byte floats, this represents a 40x space reduction.

3.3 Searching

Searching in Asymmetric Hashing works by calculating approximate distances between the target and database vectors. It does so by computing exact distances between the target vector and the approximately reconstructed database vector:

$$d'(u, v^i) = d(u, Approx(v^i)) \approx d(u, v^i) \tag{14}$$

The ANN search happens in two steps. First, we construct a lookup table of distances. Then we use that table to quickly compute approximate distances to all vectors. This process is explained in detail below.

Given a search vector u, we construct a $\frac{N}{k}$x256 matrix of distances to each centroid:

$$D_{i,j}(u) = d(\pi_i(u), c_i^j) = \left|\pi_i(u) - c_i^j\right|^2 \quad (15)$$

Assuming that we use 4 byte floats, this requires $\frac{N}{k}$ kilobytes of memory. For $N = 400$ and $k = 10$, this is a very small amount of storage, approximately 40 kilobytes.

Given this lookup table, we can group terms to compute approximate distances via lookups and additions. Here we set $A = Approx(v^i)$ for conciseness:

$$\begin{aligned}
d'(u, v^{(i)}) &= d(u, Approx(v^{(i)})) \\
&= d(u, A) \\
&= (u_1 - A_1)^2 + (u_2 - A_2)^2 + \ldots + (u_N - A_N)^2 \\
&= [(u_1 - A_1)^2 + (u_2 - A_2)^2 + \ldots + (u_k - A_k)^2] + \ldots + \\
&[(u_{N-k+1} - A_{N-k+1})^2 + (u_{N-k+2} - A_{N-k+2})^2 + \\
&\ldots + (u_N - A_N)^2] \\
&= |\pi_1(u) - \pi_1(A)|^2 + \ldots + \left|\pi_{N/k}(u) - \pi_{N/k}(A)\right|^2 \\
&= \left|\pi_1(u) - c_1^{h_1(v^{(i)})}\right|^2 + \ldots + \left|\pi_{N/k}(u) - c_{N/k}^{h_{N/k}(v^{(i)})}\right|^2 \\
&= D_{1,h_1(v)} + D_{2,h_2(v)} + \ldots + D_{N/k,h_{N/k}(v)}
\end{aligned}$$

So computing an approximate distance requires $\frac{N}{k}$ lookups and $\frac{N}{k} - 1$ additions. Given a database V of M vectors and the lookup table, we can compute the approximate distance from u to all vectors in V using $\frac{MN}{k}$ additions. Since we typically have $\frac{M}{k} \gg 256$, the main loop is the performance bottleneck.

This derivation depends on the fact that square Euclidean distance can be summed across the chunks. There is no requirement that a distance function have this property, and many do not. In particular, Pearson correlation does not have this property when the vectors have not been appropriately normalized.

We can make use of this summing property to add an important optimization: the "early-out". Since the distance is monotonically increasing as we add chunks, once a particular vector has accumulated a large enough distance to establish that it is not one of the nearest neighbors, that vector no longer needs to be processed. This optimization becomes more effective as M, the number of vectors on each machine, increases. This somewhat mitigates the advantages of sharding across many machines.

3.4 Intuition

The "Asymmetric" in "Asymmetric Hashing" refers to the fact that we hash the database vectors but not the search vectors. The approximate distance function takes a vector and a hash code as inputs, so it cannot be symmetric.

Unlike algorithms involving locality-sensitive hashing, Asymmetric Hashing only hashes the database vector, thus eliminating one source of error. An illustration of this reduced error can be seen in Figure 3.

3.5 Second Pass Reorder

If the original vectors can be stored in memory, then it is possible to add a second pass to the Asymmetric Hashing algorithm which does an exact reorder of the top R approximate results. This can greatly increase the recall of the approximate search.

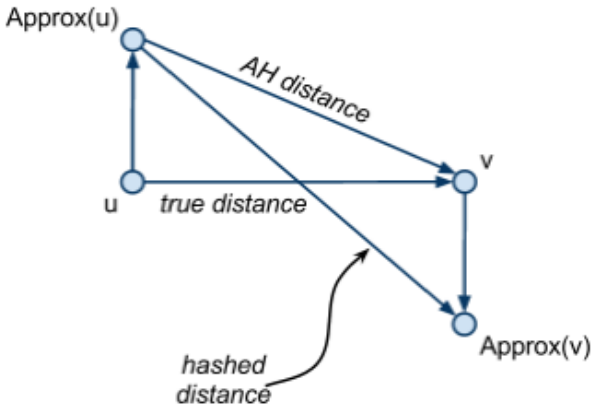

Figure 3: 2 Dimensional visualization of the distance to approximations of vectors.

A second pass reorder becomes more appealing when the search is sharded across hundreds of machines. If each of 100 machines does an exact reorder on its top 10 approximate results, then this is similar to doing an exact reorder on the top 1000 approximate results (it will not be exactly the same, since the top 1000 approximate results are unlikely to be sharded evenly). If each machine has to reorder very few results, then it may be possible to store the original features on disk, rather than in memory.

Google Correlate uses $R = 100$ and $O(100)$ machines to achieve an effective reordering of the top 10,000 approximate results on each query.

3.6 Asymmetric Hashing for Vectors with Missing Indices

This technique does not translate directly to Pearson correlation search with holdouts/missing indices, because we are unable to correctly normalize the database vectors before the search. This is because we do not know which indices are to be considered until the search vector arrives. The database vectors would have to be renormalized with every search. The essential problem is that Pearson correlation distance cannot be summed across chunks in the way that squared Euclidean distance can.

Examining the formula for Pearson correlation, as defined in (2), we can expand the numerator and denominator to give:

$$r(u, v) = \frac{\sum_{i=1}^{n} [u_i v_i - u_i \mu(v) - v_i \mu(u) - \mu(u)\mu(v)]}{\sqrt{\sum_{i=1}^{n} (u_i - \mu(u))^2} \sqrt{\sum_{i=1}^{n} (v_i - \mu(v))^2}} \quad (16)$$

$$= \frac{\sum u_i v_i - n\mu(u)\mu(v)}{\sqrt{n \sum u_i^2 - (\sum u_i)^2} \sqrt{n \sum v_i^2 - (\sum v_i)^2}} \quad (17)$$

The sums here are taken over the non-missing indices. Using(17), we can break Pearson correlation into six parts which *can* be summed across chunks. If we take

$$\begin{aligned}
n &= \text{reduced dimensionality}, & S_u &= \sum u_i \\
S_v &= \sum v_i, & S_{uu} &= \sum u_i^2 \\
S_{uv} &= \sum u_i v_i, & S_{vv} &= \sum v_i^2
\end{aligned}$$

Then:

$$d_p(u,v) = 1 - \frac{nS_{uv} - S_u S_v}{\sqrt{(nS_{uu} - S_u^2)(nS_{vv} - S_v^2)}} \quad (18)$$

To use this formula with Asymmetric Hashing, we train and index the data as before. At query time, we create lookup tables and sum each of these six quantities, rather than $|u-v|^2$. While creating the lookup tables, we skip over missing indices. We can then use (18) to compute the exact Pearson correlation distance between the vector u with missing indices (u,m) and a hashed approximation of a database vector $(h(v),\{\})$:

$$d_p((u,m),(h(v),\{\})) \approx d_p((u,m),(v,\{\}))$$

This is possible because database vectors do not have any missing indices. Since we are creating lookup tables for more quantities and summing more of them, we expect that a Pearson AH search will be slower than a conventional AH search. We quantify this slowdown later in the paper.

3.7 Theoretical Performance

We can also analyze AH in terms of Floating Point Operations (FLOPs). We count addition, subtraction and multiplication as single operations.

The AH algorithms for Euclidean distance and for Pearson Correlation with missing indices both operate in three steps: the construction of a lookup table, the calculation of approximate distances using that table and, finally, a reordering of the top candidates using exact distances.

3.7.1 Squared Euclidean distance

Here we calculate the number of FLOPs required to run the AH algorithm for "full" vectors which have been pre-normalized.

Lookup Table: for each of the $256(\frac{N}{k})$ centroids, we must compute $(a-b)^2$ for each of the k dimensions, and then add the results. This requires $256\frac{N}{k}(2k+(k-1)) = 256N(3-\frac{1}{k})$ operations.

Approximate Distances: for each vector we look up the pre-computed distances for each chunk and add them. There are M vectors and $\frac{N}{k}$ chunks, so this requires $M(\frac{N}{k}-1)$ additions.

Reorder: For each of the R vectors to be reordered, we must compute $(a-b)^2$ for each of the N dimensions, then add the results. This requires $R(2N+N-1) = R(3N-1)$ operations.

So in total, discarding lower-order terms, we expect the cost of an asymmetric hashing search to be:

$$Cost = 3(256)N + M\frac{N}{k} + 3RN$$

In our case, the predicted slowdown for $N = 10^5, k = 10, R = 1000$ is 1.28.

3.7.2 Pearson Distance

When computing approximate Pearson correlation distance, we have to track five quantities rather than a single quantity. This increases costs across the board.

Assume u is the query vector and v is a database vector.

Lookup table: We can compute S_u and S_{uu} once with $N(3-\frac{2}{k})$ operations. We cannot compute S_v and S_{vv} offline, since we do not know in advance which indices will be missing in u. Computing these two quantities for all centroids requires $256N(4-\frac{2}{k})$ operations. Finally, we must compute S_{uv} at a cost of $256N(2-\frac{1}{k})$. Dropping lower-order terms, the cost of creating the lookup table is approximately $256(5N)$.

Approximate Distances: Since S_u and S_{uu} were precomputed, we need not recompute them here. The cost of computing the approximate distance is the cost of summing S_v, S_{vv} and S_{uv}, plus the cost of the Pearson correlation formula. In most cases, the cost of computing the formula is small compared to the summing. Dropping that term, we get $3M\frac{N}{k}$ operations.

Reorder: S_u and $S_u u$ are already exact, so they need not be recomputed. We must compute S_v, S_{vv} and S_{uv} exactly for each query which is to be re-ordered, then use them to compute the distance. Calculating S_v, S_{vv} and S_{uv} requires $5N-3$ operations, which is typically large compared to the cost of computing the distance. Hence this step requires $5RN$ operations.

So the overall cost is:

$$\text{Cost} = 5(256N) + 3M\frac{N}{k} + 5RN$$

Since $M \gg N$, the middle term is typically the dominant one. This indicates that we should expect to see a roughly 3x slowdown when using AH with Pearson correlation instead of ℓ^2 distance.

4. METHODOLOGY

To test the performance of the approximate algorithms, we collected a random sample of 100,000 of our database vectors. Each vector consisted of 400 weeks of data, so that there were 40M values in the test set. We stored these as 4-byte floats, so the test set required 160MB of memory.

To test full-vector searches, we ran a batch process to compute the exact 100 nearest neighbors for each of the 100,000 vectors. We built AH data using 40-10 and 20-20 configurations (i.e. $N/k = 40$, $k = 10$ and $N/k = 20$, $k = 20$). In order to ensure consistency across test runs, we limited our testing to a single threaded environment.

To test searches with holdouts, we created four different test sets from the 100,000 vectors by removing indices in different ways, as shown in figure 4:

1. The *chop* set removes W weeks from either the beginning or the end of the vector. This mirrors a data set which only became available partway through the Correlate time period.

2. The *even* set removes weeks at evenly-spaced intervals. This mirrors a system which may be shut off during an off-season.

3. The *holdout* set removes a random span of W weeks. This simulates an explicitly chosen holdout season.

4. The *spike* set removes W weeks surrounding the largest value in the time series. The model for this is a query like *Michael Jackson*, which had a 20 standard deviation spike in the week of his death. This spike drowns out all other features of the time series, which may be interesting in their own right.

The value of W (the number of weeks held out) was chosen randomly between 20 and 200 for each vector and index

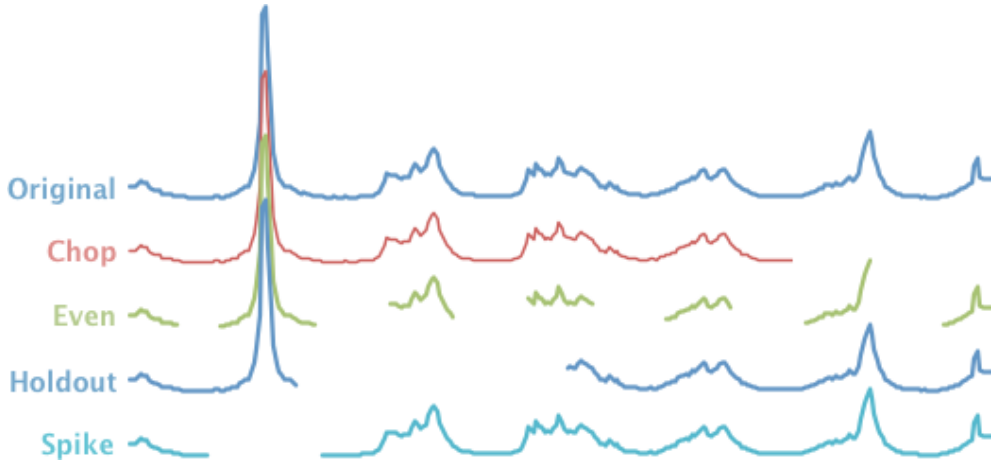

Figure 4: Time series representations for the vector for *Michael Jackson*

removal technique. Each of these test sets contained 100,000 vectors.

We measured the differences between the optimal results for each of these four sets and the original vector. We also measured the performance and recall on all four of these sets vs. an exact, brute force calculation of Pearson correlation with missing indices.

The experiments were run on a single machine. The nearest neighbor algorithm ran in a server and the performance testing binary sent RPCs to localhost. To verify that this did not introduce significant bottlenecks, we ran a server that always returned the same results. The performance testing binary reported greater than 1000 QPS. This is an order of magnitude greater than the QPS from our real system, so the communication and testing overhead are not significant.

5. RESULTS

Technique	Median	Mean	Std. Dev.
chop	54	51.27	30.19
even	67	63.70	21.29
holdout	66	60.28	26.60
spike	31	35.04	26.72

Table 1: Recall@100 between optimal results for "full" vectors and the same vectors with missing indices added using different index removal techniques.

Table 1 shows statistics on the size of the intersection between the full-vector top 100 exact nearest neighbors and the missing indices top 100 exact nearest neighbors for each index removal technique. The sample size is 100,000 vectors.

The *even* and *holdout* techniques changed the exact results the least, moving 35-40 neighbors out of the top 100 on average. The *spike* technique changed the results the most, removing 65 of the top 100 results for the full vector on average. This is to be expected, since the largest value (which the *spike* technique removes) has the greatest influence on correlation values.

Table 2 shows the results for full vector search. Asymmetric Hashing is able to achieve a 15x speedup over brute force search, but at the cost of poor recall (46.55%). Introducing an exact reordering step increases the recall to 97.54% while slightly reducing the throughput (only a 10.2x speedup is achieved).

The theoretical prediction for the slowdown of an $R = 1000$ reorder step with a 40-10 configuration was 1.28, while the observed slowdown is 1.45. The predicted slowdown for $N = 10^5, k = 20, R = 1000$ is 1.52, while observed slowdown is 1.62.

The explanation for the difference is the early-out:

1. The average approximate distance calculation requires fewer than N/k additions because many vectors trigger the early-out condition. So the main loop actually takes less time than the FLOP calculations indicate.

2. For large R, more vectors must be tracked as candidates. If there is no reorder and we wish to find the top ten results, we need only track the ten best candidates seen so far. The tenth candidate defines the early-out cutoff. But if $R = 1000$, we must track the 1,000 best candidates and the 1,000th defines the (much looser) cutoff.

Table 3 presents the results for vectors with missing indices. The results are analogous to those for full-vector searches, but with a lower speedup. This is to be expected, since there is more data being pre-computed and summed. AH achieves a 4.5x speedup with approximately 50% recall. Adding a 200-element second-pass reorder increases recall to 97% for most data sets with minimal loss of throughput.

The theoretical model predicted a 3x slowdown for the AH/Pearson combination as compared to standard Euclidean AH. In reality, we see a 5x slowdown. This is again explained by the early-out. This optimization cannot be performed with Pearson correlation. It makes the main loop of approximate L2 search run faster in reality than the theoretical $M\frac{N}{k}$.

We do see some variation across the data sets. The *chop* and *even* sets tend to have the highest recall, while *spike* always has the lowest. This makes some sense: the *spike* set is more likely to remove the largest features from a chunk, thus making the centroids on that chunk poor representations of their original vectors.

It might be objected that the brute force algorithms are adequate, since our goal at the outset was to return results in

under 200ms. Exact search responds in 63ms for "full" vectors and 100ms for vectors with holdouts. While this is true for the experimental setup presented in this paper, the performance boosts of Asymmetric Hashing are still valuable. They allow us to put more vectors on each machine, thus reducing the overall resource requirements of our product. This has made it feasible to expand the system to countries beyond the United States.

6. CONCLUSIONS

By using Asymmetric Hashing with a second-pass reorder and modifications to support Pearson correlation with holdouts, we are able to achieve all the requirements of Google Correlate: fast searches, high-recall results and support for holdouts. We achieve 10x speedups on full-vector searches (the common case) and 97% recall. We achieve a 4x speedup on holdout searches, also with 97% recall. This search algorithm allows us to serve the requests of Correlate users with low latency.

7. ACKNOWLEDGMENTS

We would like to thank Matt Mohebbi and Julia Kodysh for their help in developing Google Correlate and Nemanja Petrovic for his help exploring Approximate Nearest Neighbor search techniques. We would also like to thank Craig Neville-Manning and Corinna Cortes for their help leading the project and reviewing this paper.

8. REFERENCES

[1] H. Choi and H. Varian. Predicting the present with google trends.

[2] A. Gersho and R. Gray. *Vector quantization and signal compression*, volume 159. Springer Netherlands, 1992.

[3] J. Ginsberg, M. Mohebbi, R. Patel, L. Brammer, M. Smolinski, and L. Brilliant. Detecting influenza epidemics using search engine query data. *Nature*, 457(7232):1012–1014, 2008.

[4] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2130–2137. Ieee, 2009.

[5] T. Liu, A. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms.

[6] T. Liu, C. Rosenberg, and H. Rowley. Clustering billions of images with large scale nearest neighbor search. In *Applications of Computer Vision, 2007. WACV'07. IEEE Workshop on*, pages 28–28. IEEE, 2007.

[7] T. Liu, C. Rosenberg, and H. Rowley. Building parallel hybrid spill trees to facilitate parallel nearest-neighbor matching operations, May 26 2009. US Patent 7,539,657.

[8] M. Mohebbi, D. Vanderkam, J. Kodysh, R. Schonberger, H. Choi, and S. Kumar. Google correlate whitepaper. 2011. Available at www.google.com/trends/correlate/whitepaper.pdf.

Table 2: Experimental result techniques with different configurations

AH Config (k-N/k)	Reorder	Recall@10	QPS	QPS/15.82	Med. Latency (ms)	Memory (MB)
exact	none	1	15.82	1.00	63.2622	269
40-10	none	0.4655	240.31	15.19	4.407	56
40-10	100	0.8641	232.54	14.70	4.588	276
40-10	200	0.9119	216.51	13.69	4.888	279
40-10	1000	0.9754	160.92	10.17	6.376	277
20-20	none	0.337	368.40	23.29	2.642	52
20-20	100	0.7134	358.42	22.66	2.749	271
20-20	200	0.7843	347.73	21.99	2.905	270
20-20	1000	0.9051	236.39	14.95	4.268	268

Table 3: Experimental result techniques with different configurations

AH Config	Reorder	Data Set	Recall@10	QPS	Med. Latency (ms)	QPS vs. baseline
exact	n/a	chop	1	9.80	100.17	1.00
exact	n/a	even	1	10.15	97.44	1.00
exact	n/a	holdout	1	9.95	99.51	1.00
exact	n/a	spike	1	9.80	100.49	1.00
40-10	0	chop	0.5389	44.28	22.62	4.52
40-10	0	even	0.5343	44.37	22.56	4.37
40-10	0	holdout	0.5183	44.42	22.54	4.46
40-10	0	spike	0.4824	44.47	22.53	4.54
40-10	100	chop	0.9385	43.12	23.20	4.40
40-10	100	even	0.9529	42.69	23.36	4.21
40-10	100	holdout	0.9352	42.97	23.27	4.32
40-10	100	spike	0.8967	43.01	23.25	4.39
40-10	200	chop	0.9681	41.65	24.00	4.25
40-10	200	even	0.9788	41.49	24.10	4.09
40-10	200	holdout	0.9685	41.08	24.30	4.13
40-10	200	spike	0.9350	41.28	24.16	4.21
40-10	1000	chop	0.9917	33.38	29.88	3.41
40-10	1000	even	0.9967	33.20	30.07	3.27
40-10	1000	holdout	0.9943	33.21	30.06	3.34
40-10	1000	spike	0.9644	33.35	29.92	3.40
20-20	0	chop	0.4026	70.74	14.06	7.22
20-20	0	even	0.3962	71.60	14.01	7.06
20-20	0	holdout	0.3806	71.42	14.01	7.17
20-20	0	spike	0.3485	71.81	13.94	7.33
20-20	100	chop	0.859	68.35	14.64	6.98
20-20	100	even	0.8708	68.11	14.68	6.71
20-20	100	holdout	0.8499	68.29	14.65	6.86
20-20	100	spike	0.7893	67.98	14.68	6.94
20-20	200	chop	0.9215	65.10	15.36	6.64
20-20	200	even	0.9369	64.86	15.42	6.39
20-20	200	holdout	0.9191	65.04	15.37	6.53
20-20	200	spike	0.8659	65.04	15.38	6.64
20-20	1000	chop	0.9785	47.23	21.12	4.82
20-20	1000	even	0.9863	46.86	21.29	4.62
20-20	1000	holdout	0.9819	47.23	21.12	4.74
20-20	1000	spike	0.9457	47.08	21.20	4.80