Computing Large-Scale Similarities : Distributed Locality Sensitive Hashing

Abstract

Many applications from various domains such as World Wide Web, Mobile, and NLP rely on finding nearest neighbors given a data instance. Due to the very large scale (e.g., users' queries from commercial search engines), computing nearest neighbors is often required to understand users' intents. However, it is a non-trivial task as the computational complexity grow significantly with the number of queries. To challenge the aforementioned problem, we exploit Locality Sensitive Hashing (a.k.a, LSH) methods and propose novel varaints in a distributed computing environment (e.g. Hadoop). The experimental results demonstrate our proposed variants of LSH achieve the robust performance with better recall compared with Vanilla LSH even when using the same space determined by the number of hash tables .

1 Introduction

Millions of users visit commercial search engines and "query" their interests. To provide users with high quality of services, search engines such as Bing, Google, and Yahoo require intelligent analysis to realize users' implicit intents, in particular taking advantage of large scaled query logs. One of the key interesting tasks involved in learning users' implicit intents often involves computing nearest neighbors (queries) for a user given query. Computing nearest neighbors is useful for many search-related problems on the Web and the Mobile such as finding related queries (Jones et al., 2006; Jain et al., 2011; Song et al., 2012), finding near-duplicate queries (), spelling correction (), , paraphrasing (Petrovic et al., 2012; Ganitkevitch et al., 2013), and diversifying search results (Song et al., 2011).

In this research, we focus on finding nearest neighbors for a given query from very large scaled query logs available from a commercial search engine. However, computing nearest neighbors of a query from large-scale query logs is a non-trivial task due to the computational complexity among hundreds of millions of queries. Note that even in a distributed computing environment such as Hadoop, the naive pairwise computation is not feasible from our experiments using 2000 random queries to attempt to retrieve neighbors from the query logs consisting of hundreds of millions of queries. In other words, even though the big O notation is denoted as linear, *2000*N* comparisons are not feasible when N reaches to hundreds of millions.

In order to address the computational challenge, this paper exploits to find nearest neighbors by doing a *small* number of comparisons – that is, sublinear in dataset size – instead of brute force linear search. In addition to *small* number of comparisons, we aim to retrieve neighboring candidates with a 100% precision and a large recall. It is important that false positive rate (ratio of "incorrectly" identifying queries as neighbors) is to be penalized more severely than false negative (ratio of missing "true" neighbors).

The methods we propose in this paper meet all these criterions by extending existing research in Locality Sensitive Hashing (Indyk and Motwani, 1998; Charikar, 2002; Andoni and Indyk, 2006; Andoni and Indyk, 2008) to novel variants. In particular, we develop the framework of the variants on a distributed system, Hadoop by taking advantage of its distributed computing power.

Our work includes following contributions:

1. We present vanilla LSH algorithm based on the seminal research of Andoni and Indyk (2008). To best of our knowledge, this is the first paper that applies this algorithm to NLP applications.

2. We propose four novel variants of vanilla LSH motivated by the research on Multi-Probe LSH (Lv et al., 2007). We show that two of our variants achieve significantly better recall than the vanilla LSH by using the same number of hash tables. The main idea behind these variants is to intelligently look up multiple buckets within a table that have a high probability of containing the nearest neighbors of a query.

3. We present a framework on Hadoop that efficiently finds nearest neighbors for a given query from a commercial large-scale query logs in sublinear time.

4. We show that the applicability of our system on two real-world applications such as finding related queries and removing duplicated queries.

2 Problem Statement

We are interested in finding out, using a batch process, a *small* set of neighbor candidates for each query such that: 1) the similarity of any query term to a neighbor candidate returned is large and within a user-specified similarity threshold (τ). 2) We return an approximate set of neighbor candidates with a 100% precision and a large recall.

A naive exact brute force algorithm to solve the above problem: 1) Compute similarity between each query and all queries in the dataset. 2) Return all the queries as neighbors, which have similarity affinity within a similarity threshold (τ).

This algorithm is exact as it has both 100% precision and recall. However, the algorithm is naive as it is not scalable for hundreds of millions of queries. The reason being conducting pairwise computations between all the queries in the dataset involves quadratic number of comparisons in the query dataset size. For hundreds of millions of queries, it means doing petabytes of comparisons; that is computationally challenging or infeasible even in a distributed setting (such as Hadoop).

The motivated behind this research is to find nearest neighbors by doing a *small* number of comparisons (sublinear in dataset size), instead of brute force linear search. In addition to *small* number of comparisons, we also want to return a set of neighbor candidates with a 100% precision and a large recall. The method we propose meet all these criterions. We do this by exploiting existing research in Locality Sensitive Hashing (a.k.a., LSH) and propose their novel variants. In particular, we develop the framework of LSH algorithms on a distributed system (e.g. Hadoop) to take advantage of its computing efficiency.

3 Approach

We describe a distributed Hadoop framework based on Locality Sensitive Hashing (LSH). First, we present vanilla LSH algorithm based on the seminal research of Andoni and Indyk (2008). To best of our knowledge, this is the first paper that applies this algorithm to NLP applications.

This algorithm improves the existing research in LSH and Point Location in Equal Balls (Indyk and Motwani, 1998; Charikar, 2002). Point Location in Equal Balls was applied for noun clustering (Ravichandran et al., 2005) and speech tasks (Jansen and Van Durme, 2011; Jansen and Van Durme, 2012). Recent prior work on new variants of Point Location in Equal Balls (Goyal et al., 2012) for distributional similarity is a special case of Andoni and Indyk's LSH algorithm.

Next, we present four new variants of vanilla LSH algorithm motivated by the research on Multi-Probe LSH (Lv et al., 2007). A significant drawback of vanilla LSH is the requirement for a large number of hash tables in order to achieve good recall in finding nearest neighbors. The goal of Multi-probe LSH is to get significantly better recall than the vanilla LSH by using the same number of hash tables.

3.1 Vanilla LSH

The LSH algorithm relies on the existence of an LSH family. Let H be a family of hash functions mapping $\mathbb{R}^D$ to some universe S. For any two query terms p, q; we chose $h \in H$ uniformly at random; and analyze the probability that $h(p) = h(q)$. The family H is called a LSH family if it satisfies the following conditions:

1. $d(p, q) \leq R$, then $Pr[h(p) = h(q)] \geq P_1$

2. $d(p,q) \geq cR$, then $Pr[h(p) = h(q)] \leq P_2$

In above conditions, d is a distance function[1], a distance threshold $R > 0$ and an approximation factor $c > 1$.

A family is generally interesting when $P_1 > P_2$. However, the difference between P_1 and P_2 can be very small. Given a family H of hash functions with parameters (R, cR, P_1, P_2), LSH algorithm is devised by amplifying the gap between the two probabilities P_1 and P_2 by concatenating several functions. In particular, LSH algorithm concatenates K hash functions: $g(q) = (h_1(q), h_2(q), \dots, h_K(q))$. The larger value of K leads to larger gap between probabilites of collision between close queries and far queries; the probabilities are P_1^K and P_2^K respectively. The advantage of this amplification is that now the algorithm is more precise meaning the probability of similar queries falling in same bucket is big; and dis-similar queries falling in same bucket is small.

In addition to the LSH algorithm being precise, it needs to have large recall. Instead of generating one set of hash table with index $g(q) = (h_1(q), h_2(q), \dots, h_K(q))$, LSH algorithm generates L tables, $g_j(q) = (h_{1,j}(q), h_{2,j}(q), \dots, h_{K,j}(q)))$; $\forall 1 \leq j \leq L$.

3.2 LSH for Cosine Similarity

In this paper, we are interested in cosine similarity, and use the LSH family defined by Charikar (2002). For two queries; $p, q \in \mathbb{R}^D$, the cosine similarity is $\left(\frac{p.q}{\|p\|\|q\|}\right)$. The LSH functions for cosine similarity is defined using sign($\alpha.p$); where $\alpha \in \mathbb{R}^D$ is a random vector. Negative sign is represented as zero and positive sign as one; hence indexes of buckets in hash tables are K-sized bit signatures. To generate α, we exploit the prior work on generating random projections (Achlioptas, 2003; Li et al., 2006) and use hash functions trick to implicitly map features ($\mathbb{R}^D$) to a set of {-1,+1}. [2] This hashing trick is important and useful as we do not need to explicitly store the huge random projection matrix of size $D \times K \times L$.

[1]In this paper, we use cosine distance.

[2]Note, we represent our queries (p and q) using only non-zero features.

Vanilla LSH has two main steps: preprocessing the data and querying the database. The pseudo code written for vanilla LSH is summarized in .

In preprocessing step, the algorithm takes input as N queries with an associated feature vectors. In our setting, feature vector is very high dimensional and sparse; that is the clicked webpages (urls) for a given query term. During preprocessing, first we generate $L \times K$ hash functions. Next, we hash all queries using these hash functions to build L hash tables with each hash table containing at most 2^K buckets. There are at most 2^K buckets because index of each bucket: g_j is K-widith bit index. Each query term is placed in their respective buckets for all L hash tables.

In querying step, for input M test queries, we retrieve all the query terms which appear in the buckets associated with M test queries. Next, compute cosine similarity between each of the retrieved terms and the input test queries. Return all those queries as neighbors which are within a similarity threshold (τ).

In this research, we have implemented the above algorithm within the distributed setting (Hadoop). In our experiments, we show that the algorithm scales to hundreds of millions of queries (see Section 4.3).

Preprocessing: Input is N queries with their respective feature vectors

- Select L functions g_j, $j = 1, 2, \ldots, L$, setting $g_j(q) = (h_{1,j}(q), h_{2,j}(q), \ldots, h_{K,j}(q)))$, where $h_{1,j}(q), h_{2,j}(q), \ldots, h_{K,j}(q)))$ are chosen at random from the LSH family.
- Construct L hash tables, $\forall 1 \leq j \leq L$. All queries with similar g_j functions ($\forall 1 \leq j \leq L$) are placed in the same bucket.

Query: Input is queries ($q = 1, 2, \ldots, M$); M is the number of test queries

- For each $j = 1, 2, \ldots, L$
  - Retrieve all the queries from the bucket with $g_j(q)$ function as the index of the bucket
  - Compute cosine similarity between query q and all the retrieved queries. Return all the queries which have similarity affinity within a similarity threshold (τ).

3.3 Reusing Hash Functions

In Section 3.2, we showed that vanilla LSH requires $L \times K$ hash functions. However generating hash functions is computationally expensive as it takes time to read all features and evaluate hash functions over all those features to generate a single bit. To minimize the number of hash functions computations, we use a trick from Andoni and Indyk (2008) in which hash functions are reused to generate L tables. K is assumed to be even and $R \approx \sqrt{L}$. We generate $f(q) = (h_1(q), h_2(q), \ldots, h_{K/2}(q)))$ of length $k/2$. Next, we define $g(q) = (f_a, f_b)$, where $1 \leq a < b \leq R$. This scheme generates $L = \frac{R(R-1)}{2}$. This scheme requires O($K\sqrt{L}$) hash functions, instead of O(KL). For rest of this paper, we use the above trick to generate L hash tables with K-widith bit bucket-ids.

Symbol	Description
N	# of query terms
D	# of features i.e. all clicked unique urls
K	# of hash functions concatenated together $g(q) = (h_1(q), h_2(q), \ldots, h_k(q)))$ to generate the index of a table
L	# of tables generated independently with $g_j(q)$ index, $\forall 1 \leq j \leq L$
F	# of bits flipped, $\forall 1 \leq j \leq L$
τ	τ threshold
Recall	fraction of similar candidates retrieved
Comparisons	Avg # of pairwise comparisons per query

Table 1: Major Notations

3.4 Multi Probe LSH

As we discussed in Section 3.3 that generating hash functions can be computationally expensive. That is exactly one of the significant drawbacks of vanilla LSH as it requires a large number of hash tables (large L) in order to achieve good recall in finding nearest neighbors. Here, We describe four new variants of vanilla LSH algorithm motivated by the research on Multi-probe LSH (Lv et al., 2007). The goal of Multi-probe LSH is to get significantly better recall than the vanilla LSH by using the same number of hash tables. The main concept behind Multi-probe LSH is to look up multiple buckets within a table that have a high probability of containing the nearest neighbors of a query. Note, eventhough vanila LSH and Multi-probe LSH both have same number of hash tables (L); Multi-probe LSH requires more number of probes as it searches in multiple buckets within a table. The main advantage of searching in multiple buckets over generating more number of tables is that it takes more memory and time to generate more tables in pre-processing.

The original Multi-probe LSH algorithm is developed for Euclidean distance. If interested in details about Multi-probe LSH algorithm for Euclidean distance, the reader is referred to (Lv et al., 2007). Here, we propose four variants of Multi-probe LSH for cosine similarity:

- **Random Flip Q:** The first variant is our baseline: first we compute the initial LSH of a query q, which gives the L bucket ids. Next, we create alternate bucket ids by taking each of the L bucket ids and then creating alternate candidate buckets by flipping a set of coordinates randomly in the LSH of just the query.

- **Random Flip B:** The second variant is another baseline similar to the previous one. Instead of just flipping the bits for only the query, here we flip bits for both the query and all the queries in the database.

- **Distance Flip Q:** The third variant is intelligent version of first variant. Instead of randomly flipping some coordinates, we devisely select a set of coordinates based on the distance of query term from the random hyperplane (hash function) that was used in to create this coordinate. The distance of query term from the random hyperplane is the absolute value which we get before applying the sign function on it (see Section 3.2).

- **Distance Flip B:** The fourth version is similar to third one, however here we flip bits for both the query and the queries in the database (intelligent version of second baseline).

4 Experiments

We evaluate our distributed large-scale approximate similarity framework by conducting several experiments on a publicly available query logs and large-scale query logs sampled from a commercial search engine.

4.1 Data

The public dataset that we demonstrate result on is adapted from the query logs of AOL (AOL-logs dataset) search engine (Pass et al., 2006). Moreover, we show results on large-scale query logs (hundreds of millions of queries) sampled from a commercial search engine. The Qlogs001 denotes 1% sampled query logs from a commercial search engine, Qlogs010 (10%), and Qlogs100 (100%). Each query point has an associated feature vector, that is high dimensional and sparse. Feature vector contains the webpages (urls) that are weighted based on click through rate on the urls for the query term. As a pre-processing step, we remove all those queries that have less than or equal to five clicked urls. Table 2 summarizes the statistics of our several query-logs datasets. Our biggest dataset has $N = 600$ million unique queries with billions of unique urls as features.

Data	N	D
AOL-logs	0.3×10^6	0.7×10^6
Qlogs001	6×10^6	66×10^6
Qlogs010	62×10^6	464×10^6
Qlogs100	617×10^6	2.4×10^9

Table 2: Query-logs statistics

τ	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
0.7		.63		.67
0.8	57	.84	1052	.81
0.9		.98		.96

Table 3: Varying τ with fixed $K = 16$ and $L = 10$ on AOL-logs and Qlogs001.

Test Data: We conduct all the experiments on 2000 random queries sampled from the query logs. For evaluation, we compute the true similar candidates for all 2000 test queries by calculating cosine similarity between 2000 test queries and all the queries in the dataset. In this paper, we set similarity threshold $\tau = 0.7$ that means the algorithm need to retrieve candidates that have cosine similarity larger than or equal to τ.

4.2 Evaluation Metrics

We use two metrics for evaluation: Recall and Comparisons. Recall is the fraction of *true* similar candidates (found in exact cosine similarity returned candidates) retrieved by approximate LSH algorithms. Comparisons is the average number of pairwise comparisons per query. The goal of this paper is to maximize recall and minimize comparisons.

4.3 Evaluating Vanilla LSH

In the first experiment, we vary the similarity threshold parameter $\tau = 0.7, 0.8, 0.9$ with fixed $K = 16$ and $L = 10$ on AOL-logs and Qlogs001 datasets. Table 3 shows that as expected finding near-duplicates (0.9) is easier than nearest neighbors (0.7). For, rest of this paper, we fix $\tau = 0.7$ as we are interested in both near-duplicates and nearest-neighbors.

L	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
1	7	.28	106	.36
10	57	.63	1052	.67
28	152	.77	2908	.78
55	297	.89	5648	.84

Table 4: Varying L with fixed $K = 16$ on AOL-logs and Qlogs001 with $\tau = 0.7$.

K	AOL-logs		Qlogs001	
	Comparisons	Recall	Comparisons	Recall
4	112,347	.98	2,29,2670	.96
8	11,008	.90	221,132	.88
16	57	.63	1,052	.67

Table 5: Varying K with fixed $L = 10$ on AOL-logs with $\tau = 0.7$.

Data	Comparisons	Recall
AOL-logs	57	.63
Qlogs001	1,052	.67
Qlogs010	10,515	.64
Qlogs100	105,126	-

Table 6: Fixed $K = 16$ and $L = 10$ on different sized datasets with $\tau = 0.7$.

Data	Comparisons	Recall
AOL-logs ($K = 16$)	57	.63
Qlogs001 ($K = 16$)	1,052	.67
Qlogs010 ($K = 20$)	695	.53
Qlogs100 ($K = 24$)	464	-

Table 7: Best parameter settings (Based on minimizing number of comparisons and maximizing recall) of K with fixed $L = 10$ on different sized datasets.

In the second experiment, we vary $L = \{1, 10, 28, 55\}$ with fixed $K = 16$ on AOL-logs and Qlogs001 datasets. L denotes the number of hash tables and K denotes the length of the index of the buckets in the table. If we increase K (increasing the precision to reduce false positives), we also need to increase L to get good recall (increasing the recall to reduce false negatives). Table 4 shows that increasing L leads to better recall, however at the expense of more comparisons on both the datasets. In addition, having large L means generating large number of random projection bits and hash tables that is both time and memory intensive. Hence, we fix $L = 10$, which leads to reasonable recall with fewer comparisons.

In the third experiment, we vary $K = \{4, 8, 16\}$ with fixed $L = 10$ on AOL-logs and Qlogs001 datasets. As expected, Table 5 shows that increasing K reduces the number of comparisons and worse recall on both the datasets. This is intuitive as the larger value of K leads to larger gap between probabilites of collision between close queries and far queries (see Section 3.1). Hence, we fix $K = 16$ to have fewer number of comparisons.

In the fourth experiment, we fix $L = 10$ and $K = 16$ and increase the size of training data. Table 6 demonstrates that as we increase the training data size, the number of comparisons also increase. This result indicates that K needs to be tuned with respect to a specific dataset, as large K will reduce the probability of dis-similar queries falling within the same bucket. K and L can be tuned by randomly sampling small set of queries.[3]

In the fifth experiment, Table 7 shows the best K and L parameter settings on different sized datasets.[4] On our biggest dataset of 600 million queries, we set $K = 24$, $L = 10$. These parameter settings require only 464 comparisons to find approximate nearest neighbors compared to exact cosine similarity that involves brute force search over all 600 million queries in the dataset.

4.4 Evaluating Multi-Probe LSH

In the first experiment, we compare flipping the bits in query only. We evaluate two approaches: Random Flip Q and Distance Flip Q. We can make several observations from Table 8: 1) As expected, increasing the number of flips improve recall at expense of more comparisons for both Distance Flip Q and Random Flip Q. 2) Our results show that Distance Flip Q has significantly better recall than Random Flip Q with similar number of comparisons. In second row of the table with $F = 2$, Distance Flip Q has nine points better recall than Random Flip Q.

In the second experiment, we compare flipping the bits in both query and the dataset. We can make

[3] In this paper, we randomly select 2000 queries to tune parameter K.

[4] Recall on Qlogs100 cannot be computed, as it was computationally intensive to find exact similar neighbors.

similar observations from Table 9 as made in the first experiment. In the second row of the Table with $F = 2$, Distance Flip B has thirteen points better recall than Random Flip B with similar number of comparisons. Comparing across second row of Table 8 and 9 shows that flipping the bits in both query and the dataset has better recall at the expense of more comparisons. This is expected as flipping both means that we can find queries at distance two (one flip in query, one flip in dataset), hence more queries in each table when we do probe. Note, we also compared distance based flipping with random flipping on different sized data-sets, and found distance based flipping is always significantly better in terms of recall as compared to random flipping.

In the third experiment, we show the results of both variants of distance based Multi Probe that is Distance Flip Q and Distance Flip B on different sized datasets. Table 10 shows the results with parameter $L = 10$, $F = 2$, and data-size dependent K (same settings of K for different sized datasets is used as in the last experiment of Section 4.3). As observed in the last experiment, flipping bits in both query and the dataset is significantly better in terms of recall (eight points on Qlogs001 and Qlogs010) with more number of comparisons.

In the fourth experiment, Table 11 shows the qualitative results for some arbitrary queries. These results are found by applying our system (Distance Flip B with parameters $L = 10$, $K = 24$, and $F = 2$) on Qlogs100. The first two columns in Table 11 show that the returned approximate similar neighbors can be useful in finding related queries (Jones et al., 2006; Jain et al., 2011). The third column shows an example, where we can find several popular spell errors. The last column shows different variants of a query, where user intends to find out the "weather in trumbull ct". Modern search engines provide a direct display with respect to "weather" related queries. Hence, if we don't have enough evidence about a specific query being related to "weather", we can use queries approximately similar to it to infer that if this query is about "weather" or not.

5 Applications

We show the applicability of our nearest neighbors geneated using the approximate Distance Flip algorithm with parameters $L = 10$, $K = 24$, and $F =$ 2 on two well-known query understanding tasks: finding related queries and removing de-duplcate queries.

Method	Random Flip Q		Distance Flip Q	
F	Comparisons	Recall	Comparisons	Recall
1	108	.65	106	.72
2	159	.66	*155*	**.75**
5	311	.70	*303*	**.79**

Table 8: Flipping the bits in the query only with $K = 16$ and $L = 10$ on AOL-logs with $\tau = 0.7$.

Method	Random Flip B		Distance Flip B	
F	Comparisons	Recall	Comparisons	Recall
1	204	.71	192	.80
2	433	.73	*405*	**.86**
5	1557	.86	1475	.93

Table 9: Flipping the bits in both the query and the dataset with $K = 16$ and $L = 10$ on AOL-logs with $\tau = 0.7$.

6 Related Work

7 Discussion and Conclusion

References

Dimitris Achlioptas. 2003. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687.

Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions.

Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of the ACM*.

Moses S. Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, STOC.

Juri Ganitkevitch, Benjamin Van Durme, and Chris Callison-Burch. 2013. Ppdb: The paraphrase

Method	Distance Flip Q		Distance Flip B	
$Data$	Comparisons	Recall	Comparisons	Recall
AOL-logs ($K = 16$)	155	.75	405	.86
Qlogs001 ($K = 16$)	2980	.76	7904	.84
Qlogs010 ($K = 20$)	1954	.64	5242	.72
Qlogs100 ($K = 24$)	1280	-	3427	-

Table 10: Best parameter settings (Based on minimizing number of comparisons and maximizing recall) of K with fixed $L = 10$ and $F = 2$ on different sized datasets with $\tau = 0.7$.

how lbs in a ton	coldwell banker baileys harbor	michaels	trumbull ct weather
how much lbs is a ton	coldwell banker sturgeon bay wi	maichaels	trumbull ct weather forecast
number of pounds in a ton	coldwell banker door county	machaels	weather in trumbull ct
how many lb are in a ton	door county wi mls listings	mechaels	weather in trumbull ct 06611
How many pounds are in a ton?	door county realtors sturgeon bay	miachaels	trumbull weather forecast
how many pounds in a ton	DOOR CTY REAL	michaeils	trumbull ct 06611
1 short ton equals how many pounds	door county coldwell banker	michaelos	trumbull weather ct
how many lbs in a ton?	door realty	michaeks	trumbull ct weather report
how many pounds in a ton?	coldwell banker door county horizons	michaeels	trumbull connecticut weather
How many pounds are in a ton	door county coldwell banker real estate	michaelas	weather 06611
how many lb in a ton	coldwell banker door county wisconsin	michae;ls	weather trumbull ct

Table 11: Sample 10 similar neighbors returned by Distance Flip B with $L = 10$, $K = 24$, and $F = 2$ on Qlogs100 dataset.

database. In *North American Chapter of the Association for Computational Linguistics (NAACL)*.

Amit Goyal, Hal Daumé III, and Raul Guerra. 2012. Fast large-scale approximate graph construction for NLP. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and and Computational Natural Language Learning*.

Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC.

Alpa Jain, Umut Ozertem, and Emre Velipasaoglu. 2011. Synthesizing high utility suggestions for rare web search queries. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*. ACM.

Aren Jansen and Benjamin Van Durme. 2011. Efficient spoken term discovery using randomized algorithms. In *Proceedings of the Automatic Speech Recognition and Understanding(ASRU)*.

Aren Jansen and Benjamin Van Durme. 2012. Indexing raw acoustic features for scalable zero resource search. In *Proceedings of the InterSpeech*.

Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner. 2006. Generating query substitutions. In *Proceedings of the 15th International Conference on World Wide Web (WWW)*. ACM.

Ping Li, Trevor J. Hastie, and Kenneth W. Church. 2006. Very sparse random projections. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 287–296. ACM.

Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases (VLDB)*.

Greg Pass, Abdur Chowdhury, and Cayley Torgeson. 2006. A picture of search. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*. ACM Press.

Sasa Petrovic, Miles Osborne, and Victor Lavrenko. 2012. Using paraphrases for improving first story detection in news and twitter. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Deepak Ravichandran, Patrick Pantel, and Eduard Hovy. 2005. Randomized algorithms and NLP: using locality sensitive hash function for high speed noun clustering.

Yang Song, Dengyong Zhou, and Li-wei He. 2011. Post-ranking query suggestion by diversifying search results. In *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*. ACM.

Yang Song, Dengyong Zhou, and Li-wei He. 2012. Query suggestion by constructing term-transition graphs. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining (WSDM)*. ACM.