# Computing Large-Scale Similarities : Distributed Locality Sensitive Hashing


## Abstract

Many Web, Mobile, and NLP applications rely on finding nearest neighbors for a given query from a commercial large-scale query logs . Due to the large scale volume of queries, computing nearest neighbors from all queries using brute force linear search is a computationally time intensive task. To challenge the aforementioned problem, we exploit existing Locality Sensitive Hashing (a.k.a, LSH) methods and propose their novel variants in distributed setting (e.g. Hadoop). Our experimental results show that couple of our new variants of LSH get significantly better recall than vanilla LSH by using the same 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 very 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 from all queries. 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 will take advantage of large-scale query logs by finding nearest neighbors (queries) for a user given query from all queries from a big query logs. Each query point also has an associated feature vector, that is very high dimensional and sparse – these are the set of webpages (urls) that get clicked for this query term.

However, computing nearest neighbors (queries) for a user given query from a commercial large-scale query logs (consisting of hundreds of millions of queries) is not scalable. The reason being conducting pairwise computations between a given query and all the queries in the dataset involves linear number of comparisons in the query dataset size. For finding neighbors 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). In our experiments, our naive Hadoop implementation could not compute neighbors of 2000 randomly sampled queries over the commercial query-logs consisting of hundreds of millions of queries.

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 (Indyk and Motwani, 1998; Charikar, 2002; Andoni and Indyk, 2006; Andoni and Indyk, 2008) 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.

Our work includes following contributions:

1. We present a distributed system (Hadoop) that can approximately find nearest neighbors for a given query from a commercial large-scale query logs in sublinear time.

2. 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.

3. 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 get 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.

4. We show that the applicability of our system on two real-world applications like finding related queries (Increasing direct display coverage is related to finding related queries) and ??.

## 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 point to a neighbor candidate returned is large and within a user-specified similarity threshold ($\tau$). 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 ($\tau$).

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.

We can not tolerate false positives. We can afford false negatives as there are many relevant queries; not retrieving all of them can be sufficient for a specific problem.

## 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 (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. The main idea 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. We present a high-level idea behind the Multi-probe LSH algorithm; for more details, the reader is re-

ferred to (Lv et al., 2007).

### 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 points $p$, $q$; 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 $Pr[h(p) = h(q)] \leq P_2$

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: $g(q) = (h_1(q), h_2(q), \ldots, h_k(q))$. 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 tables with index $g(q) = (h_1(q), h_2(q), \ldots, h_k(q))$, LSH algorithm generates $L$ tables, $g_j(q) = (h_{1,j}(q), h_{2,j}(q), \ldots, h_{k,j}(q)))$; $\forall 1 \leq j \leq L$.

The pseudo code written in distributed framework (Hadoop) for vanilla LSH is shown below:

**MAP: Input: $N$ 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.

**REDUCE: Input: 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 ($\tau$).

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 $\alpha$, 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}. [1] This hashing trick is important and useful as we do not need to explicitly store the huge random projection matrix of size $D \times K \times L$.

To minimize the number of hash functions computations (time intensive to read all features and evaluate hash functions to generate a single bit), we use a trick from Andoni and Indyk (2008) in which hash functions are reused to generate $L$ tables. $K$ vis 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($KL$).

### 3.2 Multi Probe LSH

One strategy that we have experimented with is the following variant of Multi-probe LSH: first we compute the initial LSH of a query q, which gives the $L$ bucket ids. We then create alternate bucket ids by

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

| Symbol | Description |
|---|---|
| $N$ | # of query points |
| $D$ | # 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, $\forall 1 \leq j \leq L$ |
| $\tau$ | $\tau$ threshold |
| Recall | fraction of similar candidates retrieved |
| Comparisons | Avg # of pairwise comparisons per query |

Table 1: Major Notations

taking each of the $L$ bucket ids and then creating alternate candidate buckets by flipping a set of coordinates in the LSH of just the query or both the query and the database of queries. We have experimented with two different methods to choose which coordinate to flip: randomly choosing a specified number of coordinates or choosing the set of coordinates based on the distance of q from the random hyperplane that was used in to create this coordinate. Random Flip LSH Random Flips

We experimented with flipping the bits in just the query and both the query and the database of all queries. 1. Generate less number of hash functions 2. Have to compute similarity over less number of data pairs

Advantages of Flipping bits compared to increasing L: 1. Increasing $L$ means generating more random projection bits that is both time and memory intensive. 2. Probing in more tables for vanilla LSH, hence we need to read data from the disk (more disk read operations.). While Flipping bits, data is already in memory, and the given data needs to be flipped. 3. Disadvantage of flipping is we need to store random projection distances to perform flipping (which takes space).

Distance Flip LSH Based on random projection distance

# 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.

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

Table 2: Query-logs statistics

## 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 (billion 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.

**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 $\tau$.

## 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 $L = \{1, 10, 28, 55\}$ with fixed $K = 16$. $L$ denotes the number of ta-

| $L$ | Comparisons | Recall |
|---|---|---|
| 1 | 7 | .28 |
| 10 | 57 | .63 |
| 28 | 152 | .77 |
| 55 | 297 | .89 |

Table 3: Varying $L$ with fixed $K = 16$ on AOL-logs with $\tau = 0.7$.

bles and $K$ denotes the length of the index of 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 3 shows that increasing $L$ leads to better recall, however at the expense of more comparisons. In addition, having large $L$ means generating large number of random projection bits that is both time and memory intensive. Hence, we fix $L = 10$, which leads to reasonable recall with fewer comparisons.

In the second experiment, we vary $K = \{4, 8, 16\}$ with fixed $L = 10$. As expected, Table 4 shows that increasing $K$ reduces the number of comparisons and worse recall. This is intuitive as we reduce the value of $K$, it leads to brute force linear search. Hence, we fix $K = 16$ to have fewer number of comparisons.

In the third experiment, we fix $L = 10$ and $K = 16$ and increase the size of training data. Table 5 demonstrates that as we increase the training data size, the number of comparisons also increase. This result indicates that $K$ and $L$ needs to be tuned with respect to a specific dataset. $K$ and $L$ can be tuned by randomly sampling small set of queries.[2]

In the fourth experiment, Table 6 shows the best $K$ and $L$ parameter settings on different sized datasets.[3] 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.

[2]In this paper, we randomly select 2000 queries to tune $K$ and $L$ parameters.

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

| $K$ | Comparisons | Recall |
|---|---|---|
| 4 | 112,347 | .98 |
| 8 | 11,008 | .90 |
| 16 | 57 | .63 |

Table 4: 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 |

Table 5: Fixed $K = 16$ and $L = 10$ on different sized datasets with $\tau = 0.7$.

### 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 7: 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 similar observations from Table 8 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 7 and 8 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

| 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 6: Best parameter settings (Based on minimizing number of comparisons and maximizing recall) of $K$ with fixed $L = 10$ on different sized datasets.

| 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 7: 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 8: Flipping the bits in both the query and the dataset with $K = 16$ and $L = 10$ on AOL-logs with $\tau = 0.7$.

| 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 9: 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 $\tau = 0.7$.

(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 9 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 10 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 10 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.

paraphrasing (Ganitkevitch et al., 2013)

## 5 Applications

## 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 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

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


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.