Fishing in the Stream: Similarity Search over Endless Data

Naama Kraus
Technion IIT
Haifa, Israel
nkraus@campus.technion.ac.il

David Carmel
Yahoo Research
Haifa, Israel
david.carmel@ymail.com

Idit Keidar
Technion IIT and Yahoo Research
Haifa, Israel
idish@ee.technion.ac.il

***Abstract*—Similarity search is the task of retrieving data items that are similar to a given query. In this paper, we introduce the time-sensitive notion of *similarity search over endless data-streams (SSDS)*, which takes into account data quality and temporal characteristics in addition to similarity. SSDS is challenging as it needs to process unbounded data, while computation resources are bounded. We propose *Stream-LSH*, a randomized SSDS algorithm that bounds the index size by retaining items according to their freshness, quality, and dynamic popularity attributes. We show that Stream-LSH increases recall when searching for similar items compared to alternative approaches using the same space capacity.**

***Keywords*-Similarity search; Stream search; Retention policy; Locality sensitive hashing; Dynamic popularity**

I. Introduction

Users today are exposed to massive volumes of information arriving in endless data streams: hundreds of millions of content items are generated daily by billions of users through widespread social media platforms [34, 35, 4]; fresh news headlines from different sources around the world are aggregated and spread by online news services [28, 17]. In this era of information explosion it has become crucial to 'fish in the stream', namely, identify stream content that will be of interest to a given user. Indeed, search and recommendation services that find such content are ubiquitously offered by major content providers [17, 18, 28, 14, 16].

A fundamental building block for search and recommendation applications is *similarity search*, an algorithmic primitive for finding similar content to a queried item [33, 7]. For example, a user reading a news item or a blog post can be offered similar items to enrich his reading experience [36]. In the context of streams, many works have observed that applications ought to take into account temporal metrics in addition to similarity [18, 28, 27, 23, 29, 35, 21, 32, 34, 31]. Nevertheless, the similarity search primitive has not been extended to handle endless data-streams. To this end, we introduce here the problem of *similarity search over data streams (SSDS)*.

In order to efficiently retrieve such content at runtime, an SSDS algorithm needs to maintain an *index* of streamed data. The challenge, however, is that the stream is unbounded, whereas physical space capacity cannot grow without bound; this limitation is particularly acute when the index resides in RAM for fast retrieval [34, 30]. A key aspect of an SSDS algorithm is therefore its *retention policy*, which continuously determines which data items to retain in the index and which to forget. The goal is to retain items that best satisfy the needs of users of stream-based applications.

We present *Stream-LSH*, an SSDS algorithm based on *Locality Sensitive Hashing (LSH)*, which is a widely used randomized similarity search technique for massive high dimensional datasets [20]. LSH builds a hash-based index with some redundancy in order to increase recall, and Stream-LSH further takes into account quality, age, and dynamic popularity in determining an item's level of redundancy.

A straightforward approach for bounding the index size is to focus on the freshest items. Thus, when indexing an endless stream, one can bound the index size by eliminating the oldest items from the index once its size exceeds a certain threshold. We refer to this retention policy as *Threshold*. Although such an approach has been effectively used for detecting new stories [34] and streaming similarity self-join [31], it is less ideal for search and recommendations, where old items are known to be valuable to users [25, 36, 10].

We suggest instead *Smooth* – a randomized retention policy which bounds the index size, by gradually eliminating index entries over time. Since there is redundancy in the index, items do not disappear from it at once. Instead, an item's representation in the index decreases with its age. As a result, Smooth finds similar items for a longer time period with a gradually decaying recall; this comes at the cost of a lower recall compared to Threshold, when searching for fresh items. We further show that Smooth exploits capacity resources more efficiently so that the average recall is larger than with Threshold.

We extend Stream-LSH to consider additional data characteristics beyond age. First, our Stream-LSH algorithm considers items' quality. For example, the quality of a social post may be defined according to the authority of the post's author, which can be based on the number of followers of this author. Stream-LSH adjusts an item's redundancy in the index based on its quality. This is in contrast to the standard LSH, which indexes the same number of copies for all items regardless of their quality. Second, we present the *DynaPop* extension to Stream-LSH, which considers items' dynamic popularity. DynaPop gets as input a stream of user interests in items, such as retweets or clickthrough information, and re-indexes in Stream-LSH items of interest; thus, it has Stream-LSH dynamically adjust items' redundancy to reflect their popularity.

We evaluate Stream-LSH on several real-world stream datasets. We extend the recall metric to consider similarity, age, quality, and popularity radii. Our results show that Smooth increases recall when searching for similar items compared to Threshold, when using the same space capacity. We show that our quality-sensitive approach is appealing for similarity search applications that handle large amounts of low quality data, such as user-generated social data [6, 9, 11], since it increases the recall of high-quality items. Finally, we show that using DynaPop, Stream-LSH is likely to find popular items that are similar to the query, while also retrieving similar items that are not highly popular albeit with lower probability. Retrieving similar items from the tail of the popularity distribution in addition to the most popular ones is beneficial for applications such as query auto-completion [8] and product recommendation [39].

II. Similarity Search over Data-Streams

Similarity Search: Similarity search is based on a *similarity function* [15], which measures the similarity between two vectors $u, v \in V$, where $V = (\mathbb{R}_0^+)^d$ is some high d-dimensional vector space. The similarity function returns a *similarity value* within the range $[0, 1]$, where 1 denotes perfect similarity, and 0 denotes no similarity.

Given a (finite) subset of vectors, $U \subseteq V$, *similarity search* [15] finds vectors $v \in U$, which are similar to a given query vector $q \in V$. A commonly used similarity function for textual data is angular similarity [34, 32] (which is closely related to cosine similarity [12, 15]).

Similarity Search over Data-Streams: SSDS considers an unbounded *item stream* $U \subseteq V$ arriving over an infinite time period, divided into discrete *ticks*. On every time tick, 0 or more new items arrive in the stream, and the *age* of a stream item is the number of time units that elapsed since its arrival. Note that each item in U appears only once at the time it is created. Each item is associated with a quality score, which is specified by a given weighting function $qual : V \to [0, 1]$. For example, according to the authority of a social post's author.

An SSDS algorithm's input consists of a *query* vector $q \in V$ and a three-dimensional radius, $(R_{sim}, R_{age}, R_{qual})$, of similarity, age, and quality radii, respectively. An *exact SSDS algorithm* returns a unique ideal result set

$$\mathit{Ideal}(q, R_{sim}, R_{age}, R_{qual}) \triangleq \{v \in U | sim(q, v) \geq R_{sim} \wedge age(v) \leq R_{age} \wedge qual(v) \geq R_{qual}\}.$$

An approximate *SSDS algorithm* A returns a subset $\mathit{Appx}(A, q, R_{sim}, R_{age}, R_{qual})$ of q's ideal result set.

Definition II.1 (recall at radius)**.** *The* recall at radius *of algorithm A for query q and radius $(R_{sim}, R_{age}, R_{qual})$ is*

$$\text{Recall}(A, R_{sim}, R_{age}, R_{qual})(q) \triangleq \frac{|\mathit{Appx}(A, q, R_{sim}, R_{age}, R_{qual})|}{|\mathit{Ideal}(q, R_{sim}, R_{age}, R_{qual})|}.$$

The recall at radius $\text{Recall}(A, R_{sim}, R_{age}, R_{qual})$ *of A is the mean recall over the query set Q.*

Dynamic popularity: We consider a second unbounded stream I which consists of items from the item stream U and arrives in parallel to U. We call I the *interest stream*. The arrival of an item at some time tick in I signals interest in the item at that point in time. Note that an item may appear multiple times in the interest stream. We capture an item's *dynamic popularity* by a weighted aggregation of the number of times it appears in the interest stream, where weights decay exponentially with time [26]: Let $t_0, \ldots, t_n$ denote time ticks since the starting time t_0, and the current time t_n. The indicator $a_i(x)$ is 1 if item x appears in the interest stream at time t_i and is 0 otherwise. A parameter $0 < \alpha < 1$ denotes the *interest decay*, which controls the weight of the interest history and is common to all items.

Definition II.2 (item popularity)**.** *The function $pop : U \to [0, 1]$ assigns a popularity score pop(x) to an item $x \in U$: pop(x)* $\triangleq (1 - \alpha) \sum_{i=0}^{n} a_i(x)\alpha^{(n-i)}$.

Given an assignment of popularity scores to items, we are interested in the retrieval of items within a popularity radius $R_{pop} \in [0, 1]$, i.e., with a popularity score that is not lower than R_{pop}. We define recall similarly to the previous definitions.

III. Stream-LSH

A. Background: Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) [22, 20] is a widely used approximate similarity search algorithm for high-dimensional spaces, with sub-linear search time complexity. LSH limits the search to vectors that are likely to be similar to the query vector instead of linearly searching over all the vectors. LSH defines a family of hash functions $\mathcal{G}$, where a hash function $g \in \mathcal{G}$ maps a vector in dimension d into a lower dimension $k << d$. In LSH, the hashes of similar vectors are likely to collide under a random selection of g from $\mathcal{G}$. Note that the larger k is, the higher the precision.

We use here a hash family for angular similarity [12], in which $g \in \mathcal{G}$ hashes a given vector into a binary vector: $g : (\mathbb{R}_0^+)^d \to \{0,1\}^k$. At a pre-processing (index building) stage, LSH randomly samples a hash function g from $\mathcal{G}$. It then constructs a hash table and assigns each input vector v into its corresponding bucket $g(v)$. At runtime, given a query vector q, LSH computes the query's hash $g(q)$, and searches for vectors that are similar to q in the bucket $g(q)$. In order to improve recall, LSH constructs L hash tables H_i, $1 \le i \le L$, each correspond to a randomly and independently selected hash function $g_i \in \mathcal{G}$, $1 \le i \le L$. Note that each indexed vector v is now replicated in L hash tables. Given a query q, LSH searches independently over L buckets $g_i(q)$ in L hash tables.

B. Stream-LSH

We present Stream-LSH in Algorithm 1. Every time tick, Stream-LSH accepts a set of newly arriving items in the item stream U, and indexes each item into its LSH buckets. Stream-LSH selects an item's initial redundancy according to its quality: it indexes the item into each bucket with a probability that equals its quality, independently of other buckets. In addition, in order to bound the index size, in each time tick, Stream-LSH eliminates items from the index according to the retention policy it uses. Note that the two operations – indexing new items and elimination of old ones – are independent, and work independently in each bucket.

Algorithm 1 Stream-LSH

```
1:  On every time tick $t$ do:
2:  foreach $H_i \in HashTables$ do
3:      ▷ Indexing new items
4:      foreach $item \in items(t)$ do
5:          ▷ Hash to bucket $B_i$
6:          $B_i \leftarrow g_i(item)$
7:          ▷ Quality-based indexing
8:          with probability $qual(item)$, $B_i$.ADD($item$)
9:      ▷ Elimination by retention policy
10:     $H_i$.ELIMINATE()
```

C. Retention Policies

Threshold: The Threshold retention policy presented in [34, 31] sets a limit T_{size} on table size. Upon a time tick, Threshold eliminates the oldest items in each table if its size limit is exceeded. Note that with Threshold, the number of copies of an item in the index does not vary with age.

Bucket: The Bucket retention policy presented in [32] sets a limit B_{size} on bucket size (rather than on table size). Upon a time tick, Bucket eliminates the oldest items in each bucket if its size limit is exceeded. Note that with Bucket, the number of copies of an item in the index varies with age, since each bucket is maintained independently. The probability of an item to be eliminated from a bucket depends on the data distribution, i.e., on the probability that newly arriving items will be mapped to that item's bucket.

Smooth: Smooth accepts as a parameter a *retention factor* p, $0 < p < 1$. Upon a time tick, Smooth selects uniformly at random a fraction $1-p$ of the items in each table, and eliminates them. This results in each item copy in each of the tables being eliminated with probability $1-p$, independently of its copies in other tables. The number of buckets an item is indexed into thus exponentially decays over time. As we show in Section IV, Smooth entails a bounded index size that is a function of p.

D. Dynamic Popularity

DynaPop extends Stream-LSH indexing procedure to dynamically re-index items based on signals of user interests, as reflected by the interest stream I. Here, an item's redundancy increases as the interest in it increases. At each time tick, DynaPop re-indexes an item that arrives in I into each of its buckets with probability $qual(x)u$ independently of other buckets; the *insertion factor*, $0 < u < 1$, is a parameter to the algorithm. Note that in this context, an item's quality may also change dynamically over time. At each time tick, the current quality value is considered.

IV. Index Size Analysis

We analyze Stream-LSH's expected index size when using Smooth with a retention factor p, and show that the index size is bounded.

We assume that a constant number of new items μ arrive at each time unit, and that their mean quality is ϕ. Consider one hash table, and denote time ticks as $t_0, \ldots, t_n$. At time t_0, Smooth stores $\mu\phi$ items in the hash table in expectation. A ratio $1-p$ of these $\mu\phi$ items are removed at every time tick, and thus the expected number of items that arrive at t_0 and survive elimination until t_n is $p^n\mu\phi$. It follows that the expected number of items in the table at any given time during the processing of an infinite stream is $\sum_{i=0}^{\infty} p^i \mu\phi = \frac{\mu\phi}{1-p}$. The retention process is performed independently in each of the L hash tables, therefore,

Proposition 1. *If μ new items with mean quality ϕ arrive at each time unit, the expected size of an index with L hash tables using Smooth with retention factor p is $\frac{\mu\phi}{1-p}L$.*

Next, assume that the arrival rate is not constant, but the number of new items that arrive at each time unit is bounded by μ^*, which is a reasonable assumption in practical systems. Further note that ϕ is bounded by 1. Thus, at most μ^* items are indexed into each hash table at each time unit. The number of items that arrive at t_0 and survive elimination until t_n is therefore bounded by $\frac{\mu^*}{1-p}L$.

V. Empirical Study

A. Methodology

External libraries: We use Apache Lucene 4.3.0 [1] search library for the indexing and retrieval infrastructure. For retrieval, we override Lucene's default similarity function by implementing angular similarity. For the LSH family of functions, we use TarsosLSH [3].

Datasets: We use Reuters RCV1 [2] news dataset and Twitter [38, 31] social dataset. In both datasets, each item is associated with a timestamp denoting its arrival time. The Reuters dataset consists of news items from August 1996 to August 1997, and the Twitter dataset consists of Tweets collected in June 2009. These datasets do not contain quality information and so we assume $qual(x) = 1$ for all items. In order to evaluate quality-sensitivity, we use a smaller Twitter dataset [5], denoted TwitterNas, consisting of a stream of Nasdaq related Tweets spanning 97 days from March 10th to June 15th 2016. TwitterNas contains number of followers of Tweets authors, which we use for assigning quality scores to Tweets (see Section V-C). In all datasets, we represent an item as a (sparse) vector whose dimension is the number of unique terms in the entire dataset, and each vector entry corresponds to a unique term, weighted according to Lucene's TF-IDF formula.

Train and test: We partition each dataset into (disjoint) train and test sets. The train set is the prefix of the item stream up to a tick that we consider to be the current time. The test set is the remainder of the dataset, which was not previously seen by the Stream-LSH algorithm. We randomly sample an evaluation set Q of 3,000 items from the test set and compute recall over Q according to the given radii. Table I summarizes the train and test statistics.

		Train		Test	
	Time unit	Items	Ticks	Items	Ticks
Reuters	Day	756,927	343	22,986	10
Twitter	10 Minutes	18,224,293	2,705	42,296	10
TwitterNas	Day	275,946	92	18,831	5

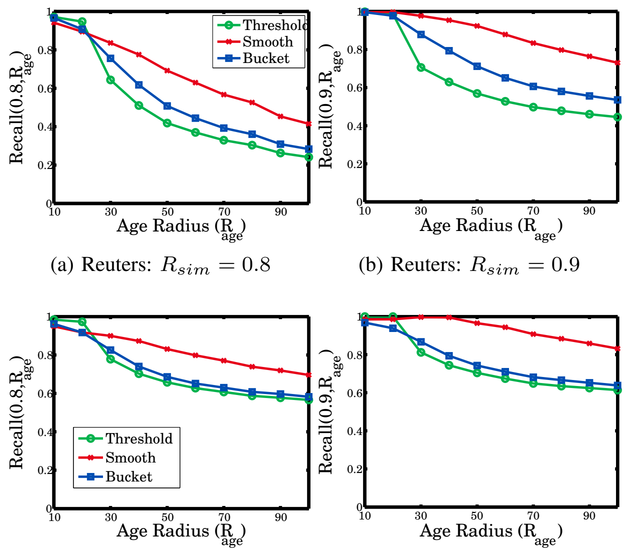
Table I: Train and test statistics.

B. Retention Policies

We evaluate the recall of the three retention policies as a function of age. As the retention aspect of our algorithm is orthogonal to the quality-sensitive indexing aspect, we assume here that $qual(x) = 1$ for all items. In order to achieve a fair comparison, we use $k = 10$ and $L = 15$ for the three retention policies, and configure them to use approximately the same index size: We set $T_{size} = 45{,}000$ and $B_{size} = 45$ in Reuters; $T_{size} = 180{,}000$ and $B_{size} = 177$ in Twitter; $p = 0.95$ in both datasets.

Figure 1 depicts recall results for Reuters in the top row, and Twitter in the bottom row. Our goal is to retrieve items that are similar to the query, hence we focus on R_{sim} values 0.8, and 0.9. As we are also interested in the retrieval of items that are not highly fresh, we evaluate recall over varying age radii values.

When considering $R_{sim} = 0.8$ (leftmost column) there is a tradeoff between Threshold and Smooth: when focusing on the highly fresh items ($R_{age} < 20$), Threshold's recall is slightly larger than Smooth's. Indeed, Threshold is effective when only the retrieval of the highly fresh items is desired. However, Smooth outperforms Threshold when the age radius increases to include also less fresh items. Bucket's recall is higher than Threshold's for ages that exceed 20, as unlike Threshold, Bucket does not eliminate items at once. Yet, Smooth outperforms Bucket when increasing the age radius due to applying an explicit gradual elimination over all items. When increasing the similarity radius to $R_{sim} = 0.9$ (leftmost column), the advantage of Smooth over Threshold becomes pronounced.



(a) Reuters: $R_{sim} = 0.8$ (b) Reuters: $R_{sim} = 0.9$

(c) Twitter: $R_{sim} = 0.8$ (d) Twitter: $R_{sim} = 0.9$

Figure 1: Recall comparison by age radius of the three retention policies using approximately the same index size.

C. Quality-Sensitivity

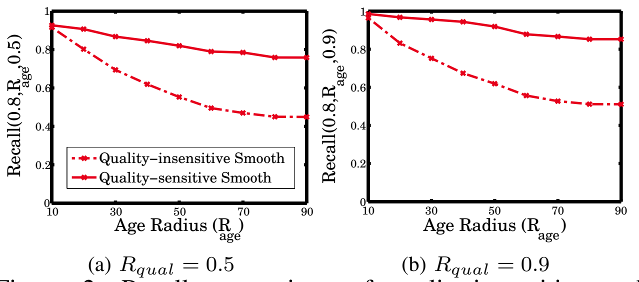
We move on to evaluating Stream-LSH's quality-sensitive approach. We experiment with the TwitterNas dataset, which contains for each Tweet x the number of followers of its author representing its authority, and denoted $T_f(x)$. We define the following quality scoring function:

$$qual(x) = log_2(1 + min(1, T_f(x)/N_f)),$$

where N_f is a configurable normalization factor. In our experiments, we set $N_f = 5{,}000$ (15% of the authors have more than 5,000 followers). Applying $qual(x)$ on TwitterNas entails an average quality score of 0.33.

We experiment with quality-sensitive and quality-insensitive variants of Smooth, with $k = 10$ and $L = 15$.

In order to conduct a fair comparison, we set retention factors that entail approximately the same index size for both variants. More specifically, we set $p = 0.9$ for the quality-insensitive variant, which results in an index size of 636,290 items in our experiment, and $p = 0.97$ for the quality-sensitive variant which results in an index size of 590,818 items in our experiment. We fix $R_{sim} = 0.8$, and experiment with $R_{qual} = 0.5$, and $R_{qual} = 0.9$ over varying age values. Figure 2 depicts the recall achieved by the two Smooth variants as a function of the age radius.



(a) $R_{qual} = 0.5$ (b) $R_{qual} = 0.9$

Figure 2: Recall comparison of quality-insensitive and quality-sensitive Smooth using approximately the same index size.

The graphs demonstrate that for both R_{qual} values, the quality-sensitive approach significantly outperforms the quality-insensitive approach when searching for similar items ($R_{sim} = 0.8$) over all age radii values that we examined. This is since the quality-sensitive approach better exploits the space resources for high quality items. The advantage of quality-sensitive indexing increases as the age of high-quality items increases, which is an advantage when the retrieval of items that are not necessarily the most fresh ones is desired. The advantage of the quality-sensitive approaches is most pronounced when there exists a large amount of low quality items in the dataset. Indeed, in our setting, 73% of the items are assigned a quality value below 0.5. In such cases, using quality-sensitive Stream-LSH is expected to be appealing for similarity-search stream applications.

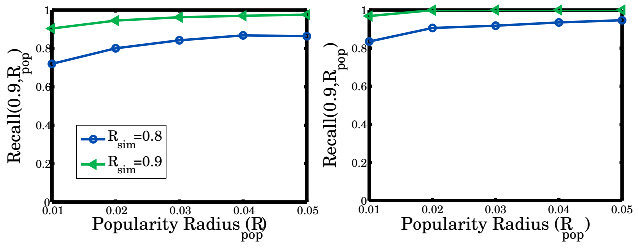
D. Dynamic Popularity

We wrap up by studying Stream-LSH when using DynaPop and the Smooth retention policy. We experiment with $u = 0.95$, $p = 0.95$. As our datasets do not contain temporal interest information, we simulate an interest stream I by considering query results as signals of interests in items [13], as follows: We use the first 75% items in the train set as the item stream U. We construct a query set Q^* by randomly sampling each item from the remaining 25% of the train set with probability 0.1. For each query $q \in Q^*$, we retrieve its top 10 most similar items in U and include them in the interest stream I at q's timestamp t_q, as well as at their original arrival times in U. Table II summarizes the item and interest stream statistics. We compute popularity scores at the current time according to Definition II.2 with $\alpha = 0.95$.

	Item stream		Interest stream	
	Items	Ticks	Items	Ticks
Reuters	540,882	252	226,890	95
Twitter	13,124,853	2,000	4,267,518	1,500

Table II: DynaPop item and interest streams statistics.

Figure 3 depicts recall as a function of R_{pop} for similarity radii 0.8 and 0.9. For both datasets and similarity radii, the recall increases as the popularity radius increases. DynaPop achieves good recall for popular items that are similar to the query while also retrieving similar items that are less popular albeit with lower recall; the latter is beneficial for applications such as query auto-completion [8] and product recommendation [39].



(a) Reuters (b) Twitter

Figure 3: Recall by popularity radius of Stream-LSH when using DynaPop with the Smooth retention policy.

VI. Related Work

Previous work on recommendation over streamed content [19, 17, 28, 24, 27, 29, 11] focused on using temporal information for increasing the relevance of recommended items. However, these search and recommendation works do not tackle the challenge of bounding the capacity of their underlying indexing data-structures. Rather, they assume an index of the entire stream with temporal information is given. Our work is thus complementary to these efforts in the sense that we offer a retention policy that may be used within their similarity search building block.

TI [13] and LSII [37] improve realtime indexing of stream data using a policy that determines which items to index online and which to defer to a later batch indexing stage. Both assume unbounded storage and are thus complementary to our work. In addition, the TI focuses on highly popular queries, whereas we also address the tail of the popularity distribution. LSII addresses the tail, however, it assumes exact search while we focus on approximate search, which is the common approach in similarity search [20].

A few previous works have addressed bounding the underlying index size in the context of stream processing [32, 34, 31, 30]. Two papers [32, 34] have focused on *first story detection*, which detects new stories that were not previously seen. Both use LSH as we do. Sundaram et al. [34] use Threshold, and Petrović et al. [32] use Bucket. These retention policies are well-suited for first story detection, however, they are less adequate in our context of similarity search as we show in our evaluation.

Morales and Gionis propose *streaming similarity self-join* (SSSJ) [31], a primitive that finds pairs of similar items within an unbounded data stream. Similarly to us, SSSJ needs to bound its underlying search index. Our work differs however in several aspects: First, we study a different search primitive, namely, similarity search. Second, SSSJ only retrieves items that are not older than a given age limit. It thus bounds the index using a variant of Threshold. In contrast, we do not assume that an age limit on all queries is known a priory. Third, we tackle approximate similarity search whereas SSSJ searches for an exact set of similar pairs.

Magdy et al. [30] propose a search solution over stream data with bounded storage, which increases the recall of tail queries. Their work differs from ours in the retrieval model, more specifically, they assume the ranking function is static and query-independent, e.g., ranking items by their age. Each item's score is known a priori for all queries, and can be used to decide at indexing time which items to retain in the index. This approach is less suitable to similarity search, where scores are query-dependent and only known at runtime.

In addition, we note that the aforementioned works on bounded-index stream processing [32, 34, 31, 30] do not take into account quality and dynamic popularity as we do.

VII. Conclusions and Future Work

We introduced the problem of similarity search over endless data-streams, which faces the challenge of indexing unbounded data. We proposed Stream-LSH, an SSDS algorithm that uses a retention policy to bound the index size. We showed that our Smooth retention policy increases recall of similar items compared to methods proposed by prior art. In addition, our Stream-LSH indexing procedure is quality-sensitive, and is extensible to dynamically retain items according to their popularity.

While our work focuses on similarity search, our approach may prove useful in future work, for addressing space constraints in other stream-based search and recommendation primitives.

Acknowledgments

Naama Kraus is grateful to the Hasso-Plattner-Institut (HPI) for the scholarship for doctoral studies. We thank Alexander Spiegelman for helpful insights and suggestions.

References

[1] Lucene. http://lucene.apache.org/core/.
[2] Reuters rcv1. http://www.daviddlewis.com/resources/testcollections/rcv1/.
[3] Tarsos-lsh. https://github.com/JorenSix/TarsosLSH.
[4] The top 20 valuable facebook statistics. https://zephoria.com/top-15-valuable-facebook-statistics/.
[5] Twitter nasdaq. http://followthehashtag.com/datasets/nasdaq-100-companies-free-twitter-dataset/.
[6] E. Agichtein, C. Castillo, D. Donato, A. Gionis, and G. Mishne. Finding high-quality content in social media. WSDM '08, pages 183–194, 2008.
[7] A. Andoni. *Nearest Neighbor Search: the Old, the New, and the Impossible*. PhD thesis, Massachusetts Institute of Technology, 2009.
[8] Z. Bar-Yossef and N. Kraus. Context-sensitive query auto-completion. WWW '11, pages 107–116, 2011.
[9] H. Becker, M. Naaman, and L. Gravano. Selecting quality twitter content for events. ICWSM11, 2011.
[10] F. R. Bentley, J. J. Kaye, D. A. Shamma, and J. A. Guerra-Gomez. The 32 days of christmas: Understanding temporal intent in image search queries. CHI '16, pages 5710–5714, 2016.
[11] M. Busch, K. Gade, B. Larson, P. Lok, S. Luckenbill, and J. Lin. Earlybird: Real-time search at twitter. ICDE '12, pages 1360–1369, 2012.
[12] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC '02*, pages 380–388, 2002.
[13] C. Chen, F. Li, B. C. Ooi, and S. Wu. Ti: An efficient indexing mechanism for real-time search on tweets. SIGMOD '11, pages 649–660, 2011.
[14] J. Chen, R. Nairn, and E. Chi. Speak little and well: Recommending conversations in online social streams. CHI '11, pages 217–226, 2011.
[15] F. Chierichetti and R. Kumar. LSH-preserving functions and their applications. In *SODA '12*, pages 1078–1094, 2012.
[16] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, G. Shen, G. Woss, C. Yang, and N. Zhang. Unicorn: A system for searching the social graph. *Proc. VLDB Endow.*, pages 1150–1161, 2013.
[17] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. WWW '07, pages 271–280, 2007.
[18] F. Diaz. Integration of news content into web results. WSDM '09, pages 182–191, 2009.
[19] E. Gabrilovich, S. Dumais, and E. Horvitz. Newsjunkie: Providing personalized newsfeeds via analysis of information novelty. WWW '04, pages 482–490, 2004.
[20] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB '99*, pages 518–529, 1999.
[21] I. Guy, T. Steier, M. Barnea, I. Ronen, and T. Daniel. Swimming against the streamz: Search and analytics over the enterprise activity stream. CIKM '12, pages 1587–1591, 2012.
[22] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. STOC '98, pages 604–613, 1998.
[23] N. Kanhabua, R. Blanco, and K. Nørvåg. Temporal information retrieval. *Foundations and Trends in Information Retrieval*, pages 91–208, 2015.
[24] M. Kompan and M. Bielikova. Content-based news recommendation. In *E-Commerce and Web Technologies*, pages 61–72. 2010.
[25] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? WWW '10, pages 591–600, 2010.
[26] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of Massive Datasets, 2nd Ed.* Cambridge University Press, 2014.
[27] L. Li, D. Wang, T. Li, D. Knox, and B. Padmanabhan. Scene: A scalable two-stage personalized news recommendation system. SIGIR '11, pages 125–134, 2011.
[28] J. Liu, P. Dolan, and E. R. Pedersen. Personalized news recommendation based on click behavior. IUI '10, pages 31–40, 2010.
[29] M. Lu, Z. Qin, Y. Cao, Z. Liu, and M. Wang. Scalable news recommendation using multi-dimensional similarity and jaccard-kmeans clustering. *J. Syst. Softw.*, pages 242–251, 2014.
[30] A. Magdy, R. Alghamdi, and M. F. Mokbel. On main-memory flushing in microblogs data management systems. ICDE '16, pages 445–456, 2016.
[31] G. D. F. Morales and A. Gionis. Streaming similarity self-join. *PVLDB*, 9(10):792–803, 2016.
[32] S. Petrović, M. Osborne, and V. Lavrenko. Streaming first story detection with application to twitter. HLT '10, pages 181–189, 2010.
[33] M. Slaney and M. Casey. Locality-sensitive hashing for finding nearest neighbors. *Signal Processing Magazine, IEEE*, pages 128–131, 2008.
[34] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *Proc. VLDB Endow.*, 6(14):1930–1941, Sept. 2013.
[35] K. Tao, F. Abel, C. Hauff, and G.-J. Houben. Twinder: A search engine for twitter streams. ICWE'12, pages 153–168, 2012.
[36] J. Teevan, D. Ramage, and M. R. Morris. #twittersearch: A comparison of microblog search and web search. WSDM '11, pages 35–44, 2011.
[37] L. Wu, W. Lin, X. Xiao, and Y. Xu. LSII: an indexing structure for exact real-time search on microblogs. ICDE '12, pages 482–493, 2013.
[38] J. Yang and J. Leskovec. Patterns of temporal variation in online media. WSDM '11, pages 177–186, 2011.
[39] H. Yin, B. Cui, J. Li, J. Yao, and C. Chen. Challenging the long tail recommendation. *Proc. VLDB Endow.*, pages 896–907, 2012.