Knowl Inf Syst
DOI 10.1007/s10115-015-0892-x

REGULAR PAPER

Continuous similarity search for evolving queries

Xiaoning Xu[1] · **Chuancong Gao**[2] · **Jian Pei**[2,3] ·
Ke Wang[2] · **Abdullah Al-Barakati**[3]

Received: 14 May 2015/Revised: 3 August 2015/Accepted: 5 October 2015

Abstract In this paper, we study a novel problem of continuous similarity search for evolving queries. Given a set of objects, each being a set or multiset of items, and a data stream, we want to continuously maintain the top-k most similar objects using the last n items in the stream as an evolving query. We show that the problem has several important applications. At the same time, the problem is challenging. We develop a filtering-based method and a hashing-based method. Our experimental results on both real data sets and synthetic data sets show that our methods are effective and efficient.

Keywords Similarity search · Data stream · Evolving query

1 Introduction

Let us consider a recommendation problem at a Q&A website. Suppose you want to run 3 ads on the Q&A website, for a user who does not have a profile yet, what ads should you display? In addition to many factors, such as click-through rates of ads and bidding price information, a natural and important idea is to consider the ads that are most related to the questions being recently asked at the website by all users. For example, technically, you may model ads and questions as sets of keywords, and may want to retrieve the top-k ads that are most similar to

This work is partly supported by an NSERC Discovery grant, the Canada Research Chair program, and a Yahoo! Faculty Research and Engagement Program (FREP) award. All opinions, findings, conclusions and recommendations in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

✉ Jian Pei
jpei@cs.sfu.ca

[1] Fortinet Inc., Burnaby, BC, Canada

[2] Simon Fraser University, Burnaby, BC, Canada

[3] King Abdulaziz University, Jeddah, Saudi Arabia

the last n questions asked, where the similarity measure captures the relevance between ads and questions. Such ads can be used as the candidates for further selections.

The above scenario is just one of the many applications that motivate the problem to be studied in this paper. Given a set of static data objects (e.g., ads in the above example), and an evolving data stream (e.g., the questions asked in the above example), a sliding window on the data stream (e.g., the last n questions in the above example) presents an evolving query. The problem of *continuous similarity search for evolving queries* is to continuously conduct top-k similarity search on the set of static objects using the evolving queries.

The problem of continuous similarity search for evolving queries has many important applications. As another example, in a computer role-playing game, a player has a set of weapons and tools, which is relatively stable. The player goes through a game mission scene, where the objects in the continuously updated surrounding environment, such as different types of enemies, scoring opportunities and obstacles, present a stream. The window over the stream containing the most recent objects approaching the player presents an evolving query. The player has to select proper weapons and tools that match the current surrounding environment best. Here, an enemy can be modeled as a set of factors, such as explosion, that it may be tackled, and a weapon/tool can be modeled as a set of factors that it may be used. Again, before any gaming strategies can be used, an essential task is to continuously maintain the top-k best weapons and tools with respect to the evolving enemies.

Our problem can be considered as the dynamic version of the top-k set similarity search problem. More generally, a traditional similarity search problem involves a collection of objects, a similarity function, and a user-defined threshold. The search is to find all objects in the collection whose similarity scores regarding the query are no less than the predefined threshold. Similarity search has many applications, such as information retrieval [14,16,33], near-duplicate web page detection [19], record linkage [37], data compression [16], data integration [9], image and video search and recommendation [13,15,31,35], statistics and data analysis [12,22], machine learning [10], and data mining [3,18]. Besides answering threshold-based queries, top-k queries are also of great value since given a threshold, the size of the result may be unpredictable and, in many real applications, we are only interested in a small number of most similar objects. Moreover, as to be reviewed in Sect. 2.2, the problem of similarity search on data streams has been extensively explored, especially for nearest neighbor search. However, to the best of our knowledge, the problem of continuous set-based similarity search for evolving queries has not been systematically investigated.

In this paper, we tackle the problem of continuous similarity search for evolving queries. Since in many applications, an object can be represented as a set or multiset, such as using a keyword vector to represent a document, we use weighted Jaccard similarity as the measure. The major challenge is how to speed up the similarity computation and avoid checking evolving queries with every static object exactly at every time point. We develop an upper bound for incremental maintenance of similarity scores. The bound can be computed in constant time. We propose two algorithms, an exact one based on the pruning and verification framework, and the other approximate one based on MinHash. We report an empirical evaluation on both synthetic and real-world data sets, which validates the efficiency and effectiveness of our proposed methods.

The rest of the paper is organized as follows. In Sect. 2, we review the related work. We then formulate the problem in Sect. 3. In sect. 4, we propose a filtering-based method. In Sect. 5, we present a hashing-based method. We report our experimental results in Sect. 6 and conclude the paper in Sect. 7.

2 Related work

Our study is mainly related to the existing work on similarity search and continuous top-k queries. In this section, we briefly review the state-of-the-art methods related to our study. A thorough survey on those topics is far beyond the capacity of the paper.

2.1 Similarity join and search

The static version of similarity search has been studied extensively. The state-of-the-art methods can be categorized into two major groups.

The first category is the filtering-based approaches. The general idea is to develop upper and lower bounds of the similarity between an object and a query, which can be computed efficiently. Many objects can be filtered out using the bounds. Consequently, the exact similarity scores, which are supposed to be more expensive to compute, are calculated for only a small number of surviving objects from filtering.

For instance, Sarawagi and Kirpal [34] proposed an inverted index-based probing method for similarity joins on sets. Chaudhuri et al. [8] developed the prefix-filtering principle for similarity joins. The all-pairs algorithm developed by Bayardo et al. [3] further improves this approach by adding the minimum length constraint and some other filtering techniques to speed up similarity joins. Nevertheless, all-pairs and prefix-filtering methods often generate a nontrivial number of candidates, which have to be verified using the exact similarity measure. Xiao et al. [38] extended the all-pairs method and proposed a new positional filtering method PPJoin, which makes use of the ordering information. PPJoin+ [38] combines suffix filtering with PPJoin and can further reduce the number of candidate pairs. A new similarity measure PathSim, which is based on meta path and is used in heterogeneous networks, is defined in [36].

The second category is hashing-based methods. The general idea is to develop hash functions that have good locality preservation properties—similar objects are likely to be hashed to the same bucket. The idea was introduced by Indyk and Motwani [21] for approximate nearest neighbor search in d-dimensional Euclidean space. The basic principle is to hash the points using multiple hash functions such that closer points have a higher probability of collision than points that are far away. Gionis et al. [17] further improved the algorithms and achieved better query time guarantees. Later, an improved algorithm that almost achieves the space and time lower bounds is presented by Andoni and Indyk [1]. The MinHash technique [5] is used to approximate the resemblance and the containment of sets. This technique is used to estimate the rarity and similarity between two windowed data streams in [11]. Moreover, Charikar [7] proposed SimHash to hash similar data to similar values. An estimation for vector-based cosine similarity using a random projection method is also discussed [7].

In this paper, we explore both filtering-based and hashing-based methods, which have not been addressed in the existing literature for evolving similarity search queries.

2.2 Continuous queries over a data stream

Different evolving models are used in previous studies that investigated continuous queries over a data stream. For example, Kontaki et al. [24] studied similarity range queries in streaming time sequences using Euclidean distance, where both the query and data objects are evolving. An indexing method that is based on incremental computation method for discrete Fourier transform is used to achieve a high candidates ratio. Lian et al. [26] tackled the similarity search problem over multiple stream time series, given a static time series as a query. An approximation algorithm is developed using a weighted locality-sensitive hashing technique.

Motivated by a wide range of applications such as network intrusion detection, much work [4,25,28,29] has been embarked on monitoring nearest neighbor (NN) queries continuously over a data stream. The basic idea is to utilize indexing structures for reducing memory consumption and supporting efficient updates. Mouratidis et al. [28] proposed two approaches for continuous monitoring of NN queries over sliding window streams. Koudas et al. [25] developed an approximation algorithm that utilizes an indexing scheme, DISC, and has guaranteed bounds on error and performance.

The existing work on continuously monitoring nearest neighbors for mobile query object is different from the problem studied here. In those previous studies, the mobile object is assumed to move in a trajectory, potentially predictable to some extent. In this paper, the stream presenting an evolving query is not assumed a moving object. Instead, we simply use the current sliding window as the current query. The existing methods on continuous nearest neighbor monitoring for mobile objects cannot solve our problem.

In addition to continuous queries on similarity search problems, some interesting problems are defined over data streams. For example, Pan and Zhu [30] developed a two-level candidate checking scheme for continuously querying the top-k correlated graphs in a data stream scenario where static queries are posed on evolving graph streams. Mouratidis et al. [27] proposed two approaches for continuously answering top-k queries where the query is a static preference function over a fixed-size sliding window. One approach is to compute new answers whenever a current top-k point expired and the other approach is to precompute future changes partially. Rao et al. [32] devised methods for the problem that uses an aggregation function to measure the relevance between a document stream and a query consists of terms. They modeled documents as a data stream. That is, new documents keep coming while the query is not evolving, which is different from our model where the evolving query consists of the elements in the current sliding window in a data stream. Kollios and Tsotras [23] proposed efficient hashing methods to answer membership queries in a temporal setting.

Table 1 summarizes the differences between our study and some previous methods related to ours.

3 Problem definition

Let us consider an alphabet of items Ψ, a collection of objects $\mathcal{R}$ where each object is a set (or multiset) over Ψ, a data stream $\mathcal{S}$ with each element $e \in \Psi$ keeps coming, and an integer n as the size of a query sliding window on $\mathcal{S}$. A query Q is the last n elements in $\mathcal{S}$. A query is in general a multiset, but also can be modeled as a set. The top-k continuous similarity search for evolving query Q is to continuously find the k objects in $\mathcal{R}$ that are most similar to Q. We answer the query continuously whenever a new element in $\mathcal{S}$ arrives.

In this paper, we consider each object being a set or multiset. Queries can be sets or multisets as well. Since a set is a special case of a multiset where elements in the object are all distinct, we use weighted Jaccard similarity to measure the similarity between objects and queries.

Specifically, let X be a nonempty multiset that consists of items in alphabet $\Psi = \{v_1, v_2, \ldots, v_{|\Psi|}\}$. We map X to a $|\Psi|$-dimensional vector $\vec{X}$ such that the value of the ith component is the absolute frequency of item v_i in X. $\vec{X}$ is called the *vector representation* of multiset X.

Table 1 Comparison of our methods and the previous work

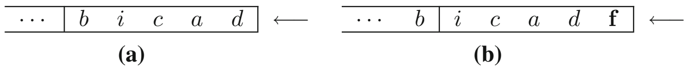
Method	Problem solved	Algorithm	Features
Our methods			
Pruning method	Top-k set/multiset similarity search	Upper bounding similarity scores w.r.t. evolving queries	Top-k, set/multiset similarity search, static objects, evolving queries
Hashing method	Top-k set similarity search	Using MinHash and inverted indices for fast computation of top-k results	Top-k, set similarity search, static objects, evolving queries
Similarity join/search—pruning-based methods			
Sarawagi and Kirpal [34]	Threshold-based set similarity join	A probing method based on inverted index	Threshold, set similarity join, static objects
Chaudhuri et al. [8]	Threshold-based set similarity join	Prefix-filtering principle	Threshold, set similarity join, static objects
Bayardo et al. [3]	Threshold-based set similarity join	All-Pairs: improved the prefix-filtering method	Threshold, set similarity join, static objects
Xiao et al. [38]	Threshold-based set similarity join	PPJoin, PPJoin+: add positional filtering and suffix filtering	Threshold, set similarity join, static objects
Sun et al. [36]	PathSim based top-k similarity search	Co-clustering based pruning method	Top-k, similarity search, static objects
Similarity join/search—hashing-based methods			
Indyk and Motwani [21]	Approximate NN search in Euclidean space	Locality-Sensitive Hashing (LSH)	Threshold, NN search in Euclidean space, static objects
Gionis et al. [17]	Approximate NN search in Euclidean space	Improved [21] and achieved better time guarantees	Threshold, NN search in Euclidean space, static objects
Andoni and Indyk [1]	Approximate NN search in Euclidean space	Almost achieves the space and time lower bounds	Threshold, NN search in Euclidean space, static objects
Broder [5]	Finding similar documents	MinHash technique	Jaccard similarity estimation, static documents
Datar and Muthukrishnan [11]	Estimate rarity and similarity of windowed data streams	Applying MinHash technique	Jaccard similarity estimation, two evolving data windows
Charikar [7]	Design new locality-sensitive hashing schemes	SimHash technique	Similarity estimation, static objects
Continuous queries over a data stream			
Kontaki et al. [24]	Similarity range queries in streaming time sequences	Indexing and Discrete Fourier Transformation	Euclidean distance, evolving objects and evolving queries
Lian et al. [26]	Similarity search over multiple stream time series	A weighted LSH	A static time series as the query, multiple stream time series as the objects
Mouratidis et al. [28]	Continuous monitoring of NN queries over sliding window streams	Conceptual partitioning and precompute the future changes	Euclidean space, evolving objects and static queries

Table 1 continued

Method	Problem solved	Algorithm	Features
Pan and Zhu [30]	Continuously querying the top-k correlated graphs in a data stream	A two-level candidate checking scheme	Evolving graph streams and static queries
Rao et al. [32]	Measure the relevance between a document stream and a query	A graph indexing structure for results sharing among queries	Evolving documents and static queries

Table 2 A collection of sets $\mathcal{R}$

T_1	a	b	c	f	i	
T_2	a	d	e			
T_3	d	e				
T_4	b	c	d	i		
T_5	a	c	g	h	i	j
T_6	a	c	e	f	h	



$\cdots$ b i c a d $\longleftarrow$
(a)
$\cdots$ b i c a d **f** $\longleftarrow$
(b)

Fig. 1 Query stream $\mathcal{S}$. **a** Current query Q, **b** query Q' after a new item arrives

Let X and Y be two nonempty multisets over alphabet Ψ. Let $\vec{X} = [x_1, x_2, \ldots, x_{|\Psi|}]$ and $\vec{Y} = [y_1, y_2, \ldots, y_{|\Psi|}]$ be the vector representations of X and Y, respectively. The weighted Jaccard similarity is

$$sim_{jac}(X, Y) = sim_{jac}(\vec{X}, \vec{Y}) = \frac{\sum_{i=1}^{|\Psi|} \min(x_i, y_i)}{\sum_{i=1}^{|\Psi|} \max(x_i, y_i)} \tag{1}$$

Example 1 (Computing similarity scores) Given an alphabet $\Psi = \{a, b, c, d, e, f, g\}$, consider two multisets $X = \{c, b, a, e, f\}$ and $Y = \{a, b, a, c, e, d\}$. The vector representations of X and Y are $\vec{x} = [1, 1, 1, 0, 1, 1, 0]$ and $\vec{y} = [2, 1, 1, 1, 1, 0, 0]$, respectively. Since $\sum_{i=1}^{7} \max(x_i, y_i) = 7$ and $\sum_{i=1}^{7} \min(x_i, y_i) = 4$, the weighted Jaccard similarity score between X and Y is $sim_{jac}(X, Y) = \frac{4}{7} = 0.57$.

We now model how a query evolves. Given a query $Q = \{e_{p+1}, e_{p+2}, \ldots, e_{p+|Q|}\}$, where item e_{p+i} $(1 \le i \le |Q|)$ is the i-th arrived item in the query, the updated query after u time instants $(u \le |Q|)$, that is, u updates in the stream, is $Q' = \{e_{p+u+1}, e_{p+u+2}, \ldots, e_{p+|Q|}, e_{p+|Q|+1}, e_{p+|Q|+2}, \ldots, e_{p+|Q|+u}\}$, where item $e_{p+|Q|+j}$ is the j-th newly coming element in the updated query, $1 \le j \le u$.

Example 2 (Continuous top-k queries) A collection of sets $\mathcal{R}$ in a database is shown in Table 2, and a data stream $\mathcal{S}$ is shown in Fig. 1a. The alphabet $\Psi = \{a, b, c, d, e, f, g, h, i, j\}$ of 10 items. Suppose $k = 2$ and the size of the sliding window is 5, that is, we use the set of the last 5 entries in stream $\mathcal{S}$ as the evolving query Q, and find the top-2 most similar sets in $\mathcal{R}$.

Table 3 Jaccard similarity scores

	T_1	T_2	T_3	T_4	T_5	T_6
Q	**0.67**	0.33	0.17	**0.80**	0.38	0.25
Q'	**0.67**	0.33	0.17	**0.50**	0.38	0.43

Table 4 Summary of frequently used symbols

Symbol	Interpretation
For all methods	
$\mathcal{R}$	A collection of objects in database
$\mathcal{S}$	The querying data stream
Ψ	The alphabet used in database
T_i	The ith object in $\mathcal{R}$
Q^t	The query containing the elements in the sliding window at time t
$\vec{X}$	The vector representation of a set (or multiset) X
$sim(X, Y)$	The exact similarity of two sets (or multisets) X and Y
k	Number of objects in the result
$topk^t$	The k objects that are most similar to the query at time t
$score(topk^t[k])$	The similarity score of the kth most similar result at time t
For pruning-based method	
$sim(X, Y)^{+u}$	The upper bound of similarity score of two sets (or multisets) X and Y, after u updates on Y
min_step_i	The least number of updates needed for object T_i's progressive upper bound $sim(T_i, Q^t)^{+min\_step_i}$ to exceed $score(topk^t[k])$
For MinHash-based method	
$\{[n]\}$	The set $\{0, \ldots, n-1\}$
$sim^{\approx}(X, Y)$	The MinHash approximated similarity of two sets X and Y
H	A set of hash functions $h_1, h_2, \ldots, h_{\lvert H\rvert}$
$index_i$	Inverted index built on the MinHash values of the ith hash function, $i \in [1, \lvert H\rvert]$
$index_i[v]$	Inverted list of hash value v which consists of the objects whose MinHash value is v according to the ith hash function, $i \in [1, \lvert H\rvert]$ and $v \in [1, \lvert\Psi\rvert]$
$min(h_i(X))$	The minimum hash (MinHash) value of set X according to the ith hash function h_i, $i \in [1, \lvert H\rvert]$
For experiments	
$\lvert\bar{Q}^t\rvert$	The average object length of the data set

We can compute the Jaccard similarity score between an object in $\mathcal{R}$ and the current query $Q = \{b, i, c, a, d\}$. The scores are shown in the first row of Table 3. Objects T_4 and T_1 are the top-2 similar objects with respect to query Q.

Suppose next an item f arrives at the data stream as shown in Fig. 1b. We now have an updated query $Q' = \{i, c, a, d, f\}$. The updated Jaccard similarity scores are shown in the last row of Table 3. The rankings of the objects in $\mathcal{R}$ ordered by similarity scores change slightly. For example, the rank of T_4 changes from 1 to 2 (the lower the rank, the higher the similarity). Consequently, T_1 and T_4 are the top-2 similar objects with respect to query Q'.

Table 4 lists the frequently used symbols in the paper.

4 A pruning-based method

Computing the exact similarity score for every object against the updated query at every time point is costly. A brute-force method takes $O(|Q^t| + |T_i|)$ time to compute the similarity between each object $T_i \in \mathcal{R}$ with the current query Q^t at every time instant t, and in total $O(|\mathcal{R}| \cdot (|Q^t| + \max_{1 \le i \le |\mathcal{R}|}\{|T_i|\}))$ time to update the top-k results at each time instant. In this section, we derive an upper bound of Jaccard similarity scores and then present a pruning-based method that utilizes the upper bound.

4.1 A progressive upper bound

We define the length of a multiset as the number of elements, including duplicates, in the multiset. To compute the upper bound of the Jaccard similarity between two multisets X and Y after u updates on Y, we can use the following property.

Theorem 1 (A progressive upper bound) *Let X and Y be two multisets, and Y' be the multiset with u updates on Y. Then,*

$$sim_{jac}(X, Y') \le sim_{jac}(X, Y)^{+u}.$$

where $sim_{jac}(X, Y)^{+u} = \frac{sim_{jac}(X,Y)\cdot\alpha+\beta}{\alpha-\beta}$ *and* $\alpha = |X| + |Y|$, $\beta = (1 + sim_{jac}(X, Y)) \cdot u$.

Proof By definition, we have

$$sim_{jac}(X, Y) = \frac{\sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}}{\sum_{i=1}^{|\Psi|} \max\{x_i, y_i\}} \tag{2}$$

and

$$\sum_{i=1}^{|\Psi|} \max\{x_i, y_i\} = |X| + |Y| - \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}. \tag{3}$$

Using Eqs. 2 and 3, we have

$$\begin{aligned}\sum_{i=1}^{|\Psi|} \min\{x_i, y_i\} &= sim_{jac}(X, Y) \cdot \sum_{i=1}^{|\Psi|} \max\{x_i, y_i\} \\ &= sim_{jac}(X, Y) \cdot (|X| + |Y| - \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\})\end{aligned}$$

Apparently, $\sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}$ can be rewritten using $|X|$, $|Y|$ and $sim_{jac}(X, Y)$ as follows.

$$\sum_{i=1}^{|\Psi|} \min\{x_i, y_i\} = \frac{(|X| + |Y|) \cdot sim_{jac}(X, Y)}{sim_{jac}(X, Y) + 1} \tag{4}$$

After u updates, the maximum increase in the intersection size (the numerator of Eq. 2) and the maximum decrease in the union size (the denominator of Eq. 2) are both u. Combining Eqs. 3 and 4, we have

$$\begin{aligned} sim_{jac}(X, Y') &\leq \frac{u + \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}}{-u + \sum_{i=1}^{|\Psi|} \max\{x_i, y_i\}} \\ &= \frac{u + \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}}{-u + |X| + |Y| - \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}} \\ &= \frac{(|X| + |Y| + u) \cdot sim_{jac}(X, Y) + u}{|X| + |Y| - u \cdot sim_{jac}(X, Y) - u} \end{aligned}$$

Let $\alpha = |X| + |Y|$ and $\beta = (1 + sim_{jac}(X, Y)) \cdot u$, we have

$$sim_{jac}(X, Y') \leq \frac{sim_{jac}(X, Y) \cdot \alpha + \beta}{\alpha - \beta} = sim_{jac}(X, Y)^{+u}$$

□

The upper bound can be computed based on $|X|$, $|Y|$, and $sim_{jac}(X, Y)$. As to be shown soon in the algorithm, $sim_{jac}(X, Y)$ is already known when computing the upper bound. Thus, it only takes constant time.

Example 3 (Progressive upper bound) Consider the same multisets X and Y in Example 1. We have $|X| = 5$, $|Y| = 6$, and $sim_{jac}(X, Y) = 0.57$. Suppose $u = 1$, the upper bound of the Jaccard similarity after an update is $sim_{jac}(X, Y)^{+1} = 0.83$.

Obviously, the progressive upper bound has a monotonicity with respect to the number of updates u. Thus, if the upper bound $sim_{jac}(X, Y)^{+u}$ is not large enough, so is any $sim_{jac}(X, Y)^{+u'}$ $(1 \leq u' < u)$.

Our strategy for deriving the upper bound after u steps can be further extended to other multiset-based similarity measures, such as the weighted cosine similarity, the weighted dice similarity, and the weighted overlap similarity. "Appendix" gives the details about the extensions to other similarity measures.

4.2 A general pruning-based algorithm

A brute-force method is to compute the similarity score between the updated query and each object at each time instant, and then obtain the top-k list. This approach is costly and involves much unnecessary computation, since when the query window slides only a small number of positions, such as 1 or 2, the changes of the similarity scores are limited.

In this subsection, we present a heuristic algorithm GP that finds the exact top-k objects continuously. For the sake of simplicity, let us assume that new elements come in a manner synchronized with time, that is, a new element arrives at the stream when there is an update in time. The main idea is based on the observation that the query at time $t + 1$ shares a large portion of elements, namely at least $\frac{n-1}{n}$, with the query at time t. Thus, the change in the top-k list is limited. In other words, the objects with small similarity scores at time t have a very limited chance to enter the top-k list at time $t + 1$.

To implement the above idea, we divide the objects into two categories according to their current similarity scores.

- The first category C_1 contains the objects whose similarity scores are small enough so that those objects will not enter the top-k list in the next several updates. The objects in this category do not need to be checked using the exact similarity scores at the next several time instants. Instead, we only need to maintain their upper bounds of the similarity scores.

– The second category C_2 contains the other objects not in the first category. The objects in this category need to be checked by computing the exact similarity scores.

Suppose that at time instant t, we obtain the top-k list of objects $topk^t$. Denote by $score(topk^t[k])$ the similarity score of the kth object, which is least similar to the current query Q^t. Apparently, $score(topk^t[k])$ can be obtained without any extra cost.

For each object T_i not in the list $topk^t$, we compute min_step_i, the smallest number of updates needed for object T_i to have its progressive upper bound exceed $score(topk^t[k])$. According to Property 1, min_step_i is the smallest integer satisfying $score(topk^t[k]) < sim(T_i, Q^t)^{+min\_step_i}$. min_step_i can be easily computed with respect to different similarity functions according to the respective specific forms of the upper bounds. For weighted Jaccard similarity, we have

$$min\_step_i = \left\lceil \frac{score(topk^t[k]) \cdot \sum_{i=1}^{|\Psi|} \max\{x_i, y_i\} - \sum_{i=1}^{|\Psi|} \min\{x_i, y_i\}}{1 + score(topk^t[k])} \right\rceil \tag{5}$$

$$= \left\lceil \frac{(score(topk^t[k]) - sim(T_i, Q^t)) \cdot (|T_i| + |Q^t|)}{(1 + score(topk^t[k])) \cdot (1 + sim(T_i, Q^t))} \right\rceil \tag{6}$$

Similarly, given Eq. 6, we only need constant time to compute min_step_i.

For the other three similarities, it is simply

$$min\_step_i = \left\lceil \frac{score(topk^t[k]) - sim(T_i, Q^t)}{sim(T_i, Q^t)^{+1} - sim(T_i, Q^t)} \right\rceil$$

Example 4 (Computing min_step_i) Again, consider the collection of sets $\mathcal{R}$ shown in Table 2 and a data stream $\mathcal{S}$ shown in Fig. 1a. As the size of the sliding window is 5, the list of the top-2 most similar objects in $\mathcal{R}$ at the current time t is $\{T_1: 0.67, T_4: 0.8\}$, where the numbers after colons are the similarity scores.

The similarity score of object T_5 at the current time instant t is 0.38. T_5 is not one of the top-2 results. However, for T_5, $\sum_{i=1}^{|\Psi|} \min\{x_i, y_i\} = 3$ and $\sum_{i=1}^{|\Psi|} \max\{x_i, y_i\} = 8$ with $x_i \in \vec{T_5}$ and $y_i \in \vec{Q^t}$. Using Eq. 5 we can compute that, after $min\_sup_5 = \left\lceil \frac{0.38 \times 8 - 3}{0.38 + 1} \right\rceil = \lceil 1.43 \rceil = 2$ updates, the progressive upper bound of T_5, $sim(T_5, Q^t)^{+2} = \frac{3+2}{8-2} = 0.83$, is larger than the score of the current kth most similar object $score(topk^t[k]) = sim(T_1, Q^t) = 0.67$.

At time instant t, for each object T_i, we maintain min_step_i and the corresponding progressive upper bound $sim(T_i, Q^t)^{+min\_step_i}$. Please note that the min_step_i values and the $sim(T_i, Q^t)^{+min\_step_i}$ values may be computed at time instant t or before. After processing the objects at time instant $t-1$, at time instant t, with respect to query Q^t, we process the objects as follows.

1. For all objects T_i such that $min\_step_i = 0$, including those already in $topk^{t-1}$, we compute the exact similarity between T_i and Q^t and obtain a top-k list $topk^t$ among those objects. Those objects are assigned to the second category C_2. We set the similarity threshold $\sigma = score(topk^t[k])$.
2. For all objects T_i such that $min\_step_i > 0$ but $sim(T_i, Q^{t-1})^{+min\_step_i} > \sigma$, since their similarity to query Q^t may be greater than $score(topk^t[k])$, we have to compute their exact similarity to Q^t, too. We update the top-k list $topk^t$ and also the similarity threshold $\sigma = score(topk^t[k])$. Obviously, in this step, $\sigma = score(topk^t[k])$ is monotonically increasing, and never decreases. At the end of this step, $topk^t$ is finalized. Those objects whose exact similarity scores are larger than $score(topk^t[k])$, that is, in the list $topk^t$, are also added to category C_2.

Algorithm 1: A general pruning algorithm (GP)

```
Input: Top-$k$ results $topk^{t-1}$ at previous time $t-1$, Query $Q^t$ at current time $t$
Output: Top-$k$ results $topk^t$ at current time $t$
1  for $0 \le u \le |queue| - 1$ do
2      foreach $T_i \in queue[u]$ do
3          if $u = 0$ or $sim(T_i, Q^t)^{+u} > score(topk^{t-1}[k])$ then
4              delete $T_i$ from bin $queue[u]$
5              compute $sim(T_i, Q^t)$ and update $topk^t$
6              $min\_step_i \leftarrow null$
7  remove bin $queue[0]$
8  foreach $T_i$ with $min\_step_i = null$ do
9      $(min\_step_i, sim(T_i, Q^t)^{+min\_step_i}) \leftarrow est\_bound\left(sim(T_i, Q^t), score(topk^t[k])\right)$
10     insert $T_i$ to bin $queue[min\_step_i]$
```

Algorithm 2: Update min_step_i and similarity bounds

```
Function: $est\_bound(sim(T_i, Q^t), score(topk^t[k]))$
Input: $T_i$'s similarity score $sim(T_i, Q^t)$ at current time $t$, The $k$th largest similarity score of top-$k$ list
        $score(topk^t[k])$ at current time $t$
Output: The pair of $(min\_step_i, sim(T_i, Q^t)^{+min\_step_i})$
1  if $sim(T_i, Q^t) < score(topk^t[k])$ then
2      compute $min\_step_i$;
3      $sim(T_i, Q^t)^{+min\_step_i} \leftarrow$ upper bound after $min\_step_i$ updates;
4  else
5      $min\_step_i \leftarrow 0$;
6      $sim(T_i, Q^t)^{+min\_step_i} \leftarrow sim(T_i, Q^t)$;
7  return $(min\_step_i, sim(T_i, Q^t)^{+min\_step_i})$;
```

3. We update min_step_i and the corresponding progressive upper bound $sim(T_i, Q^{t-1})^{+min\_step_i}$ for objects $T_i \in C_2$.
4. For the other objects not in C_2, they belong to the first category C_1 and have no hope to be more similar to Q^t than any of the object in the list $topk^t$ and thus do not need to be checked in this round. We only need to decrease their min_step_i values by 1.

The pseudocode of the algorithm is given in Algorithms 1 and 2. In implementation, we organize objects in bins according to their min_step_i values and use a heap to maintain the top-k list. All the bins are maintained in a queue structure (called $queue$ in the pseudocode).

5 A MinHash-based method

In this section, we use the technique MinHash [7], a locality-sensitive hashing (LSH) scheme, to approximate Jaccard similarity. We design indices for efficient updating estimated similarity scores. Since the MinHash technique is designed for estimating sets rather than multisets, we limit the definition of the current query to the set of distinct elements.

5.1 Approximating Jaccard similarity over static query

We first discuss how to estimate Jaccard similarity using MinHash for static objects and queries.

Definition 1 (*Image under permutation* [2]) Denote by $\{[n]\}$ the set $\{0, \ldots, n-1\}$. A permutation π on $\{[n]\}$ is a bijective function (one-to-one correspondence) from $\{[n]\}$ to itself. If $x \in \{[n]\}$, then $\pi(x)$, the value of π when applied to x is the image of x under π. The image of a subset $X \subseteq [n]$ under π is $\pi[X] = \{y \mid y = \pi(x) \text{ for } x \in X\}$.

Definition 2 (*Min-wise independent permutations* [6]) Let S_n be the set of all permutations of $\{[n]\}$. A family of permutations $\mathcal{F} \subseteq S_n$ is min-wise independent if for any set $X \subseteq \{[n]\}$ and any element $x \in X$, we have

$$Pr(\min\{\pi[X]\} = \pi(x)) = \frac{1}{|X|} \tag{7}$$

where the permutation π is chosen randomly in $\mathcal{F}$.

The following property enables efficient estimation of Jaccard similarity between two sets.

Theorem 2 (Jaccard similarity estimation [20]) *Consider a query Q and an object T_i where the elements are drawn from an alphabet Ψ. Let h be a hash function that maps elements in Ψ to distinct integers in range* $[0, |\Psi|-1]$ *and is randomly picked from a family of min-wise independent permutations. Let the MinHash value,* $\min(h(T_i))$, *be the element in T_i with the smallest hash value. Then,* $Pr[\min(h(T_i)) = \min(h(Q))] = \frac{|T_i \cap Q|}{|T_i \cup Q|}$.

The MinHash values for all objects and that of the query can be stored in a MinHash signature matrix, M, where each entry $M(i, j)$ is the MinHash value of the jth itemset under hash function h_i. Based on Property 2, the Jaccard similarity between two sets can be estimated by the ratio of the number of rows containing the same MinHash values to the number of all the rows in the signature matrix. We deliberate the computation in Example 5.

Example 5 (Jaccard similarity estimation) Consider the matrix representation of an example with four objects and a static query Q shown in Table 5a. Each column in the matrix represents a set and an element is in the set if the corresponding entry is 1. We apply five random permutations defined in Table 5b as the min-wise independent hash functions on objects and the query. The signature matrix is shown in Table 5c. For example, since object T_1 contains elements a, b, and c, the corresponding hash values according to h_4 are 1, 3, and 4, respectively. The MinHash value of T_1 according to h_4 is a since a is the element in T_1 with the smallest hash value.

We can estimate the Jaccard similarity score between two sets by computing the ratio of the number of rows containing the same MinHash values to the number of random permutations. For example, the MinHash values of T_1 and Q are the same according to random permutations h_2 and h_3. Thus, the estimated Jaccard similarity between T_1 and Q is $\frac{2}{5}$. We compare the estimated Jaccard similarity with the exact Jaccard similarity for each object in Table 5d. When objects are ordered in descending order of estimated similarity score regarding to the query, we have $T_3 > T_1 = T_4 > T_2$, which is identical to the order using exact similarity. Therefore, in this example, we can get the top-k result accurately using estimated scores.

5.2 Approximating Jaccard similarity over evolving query

To answer evolving queries, given a fixed set of hash functions, the MinHash values for all the objects are fixed and only the MinHash values for the query are subject to modification. That is, we only need to update the MinHash values for the query. Therefore, we have a fast solution shown in Algorithm 3.

Table 5 An example of Jaccard similarity estimation

Element	T_1	T_2	T_3	T_4	Q
(a) Matrix representation of sets					
a	1	1	0	1	0
b	1	0	1	0	1
c	1	0	1	0	1
d	0	1	0	1	1
e	0	0	1	1	1

Element	h_1	h_2	h_3	h_4	h_5
(b) Random permutations					
a	1	2	3	1	0
b	2	3	0	3	4
c	3	0	1	4	3
d	4	4	2	0	1
e	0	1	4	2	2

	T_1	T_2	T_3	T_4	Q
(c) Signature matrix					
h_1	a	a	e	e	e
h_2	c	a	c	e	c
h_3	b	d	b	d	b
h_4	a	d	e	d	d
h_5	a	a	e	a	d

	T_1	T_2	T_3	T_4
(d) Exact and approximated Jaccard similarity				
$sim_{jac}(T_i, Q)$	0.40	0.25	0.75	0.40
$sim_{jac}^{\approx}(T_i, Q)$	0.40	0.20	0.60	0.40

Algorithm 3 first computes the MinHash signature for an updated query. To determine whether we need to update the estimated Jaccard similarity of object T_j with respect to the updated query Q^t according to a hash function h_i, we consider the following 4 cases based on whether the MinHash values of the object and the queries are equal or not.

- Case 1: If $min(h_i(T_j)) \neq min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) = min(h_i(Q^t))$, the estimated Jaccard similarity between T_j and the updated query should be increased by $\frac{1}{|H|}$, where $|H|$ is the number of hash functions applied.
- Case 2: If $min(h_i(T_j)) = min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) \neq min(h_i(Q^t))$, the estimated Jaccard similarity between T_j and the updated query should be decreased by $\frac{1}{|H|}$.
- Case 3: If $min(h_i(T_j)) = min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) = min(h_i(Q^t))$, the estimated Jaccard similarity between T_j and the updated query remains.
- Case 4: If $min(h_i(T_j)) \neq min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) \neq min(h_i(Q^t))$, the estimated Jaccard similarity between T_j and the updated query remains, too.

We can further reduce the computational cost using inverted indices. The MinHash signature matrix that stores the MinHash values of objects can be transformed into $|H|$ inverted

Algorithm 3: A MinHash-based algorithm (MHB)

```
Input: Current query $Q^t$, $|H|$ hash functions, a set of previous MinHash values $\{min(h_i(Q^{t-1}))\}$,
       MinHash table for $|\mathcal{R}|$ objects.
Output: Top-$k$ results $topk_t$ at current time $t$
1 compute $\{min(h_i(Q^t))\}$;
2 foreach $j \in \{1, \ldots, |\mathcal{R}|\}$ do
3     foreach $i \in \{1, \ldots, |H|\}$ do
4         if $min(h_i(T_j)) \neq min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) = min(h_i(Q^t))$ then
5             $sim^{\approx}(T_j, Q^t) \leftarrow sim(T_j, Q^t) + \frac{1}{|H|}$;
6         else if $min(h_i(T_j)) = min(h_i(Q^{t-1}))$ and $min(h_i(T_j)) \neq min(h_i(Q^t))$ then
7             $sim^{\approx}(T_j, Q^t) \leftarrow sim(T_j, Q^t) - \frac{1}{|H|}$;
8     if $T_j \notin topk^t$ and $sim^{\approx}(T_j, Q^t) > score(topk^t[k])$ then
9         update $topk^t$;
```

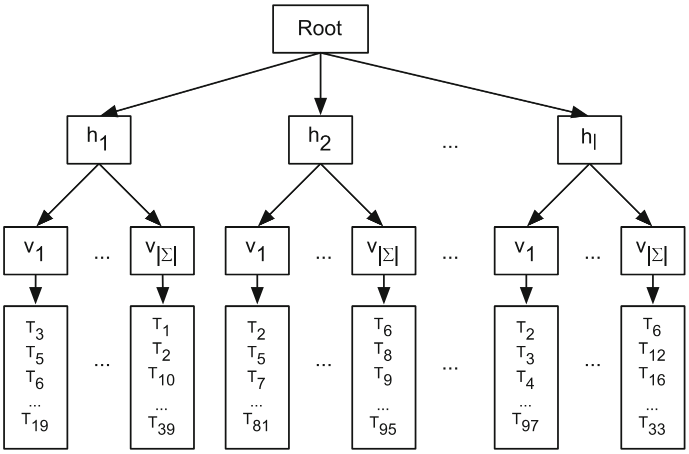


Fig. 2 Inverted indices for MinHash values

indices whose structure is shown in Fig. 2. For each hash function, its corresponding inverted index stores a mapping from each existing MinHash value to a list of object ids, that is, inverted list. Moreover, elements in each inverted list are sorted in ascending order of object id, which enables more efficient list merge and intersection operations. In this way, we can efficiently retrieve the objects with a specified MinHash value according to a certain hash function.

The algorithm that uses the inverted indices is presented in Algorithm 4. The MinHash signature for the updated query is computed first. We need to search the inverted index of a hash function h_i only when the MinHash values of Q^t and Q^{t-1} regarding h_i are different. Suppose the MinHash value of the updated query changes according to h_i, to determine the objects whose similarity scores need to be updated, we recall Case 1 and Case 2 mentioned earlier in this section. Each case corresponds to a scan in an inverted list of h_i. Specifically, in Case 1, we increase the estimated Jaccard similarity by $\frac{1}{|H|}$ for objects in the inverted list of hash value $min(h_i(Q^t))$. That is, instead of checking for each object as in Algorithm 3, we search for the objects that satisfy Case 1 using the index. Similarly, in Case 2, we decrease the estimated Jaccard similarity by $\frac{1}{|H|}$ for objects in the inverted list of hash value $min(h_i(Q^{t-1}))$.

Algorithm 4: A MinHash-based algorithm using inverted indices (MHI)

```
Input: Current query $Q^t$, $|H|$ hash functions, a set of MinHash values $\{min(h_i(Q^{t-1}))\}$, Inverted
       indices.
Output: Top-$k$ results $topk_t$ at current time $t$
1  compute $\{min(h_i(Q^t))\}$;
2  foreach $i \in \{1, \ldots, |H|\}$ do
3      if $min(h_i(Q^{t-1})) \neq min(h_i(Q^t))$ then
4          for $T_j \in index_i[min(h_i(Q^t))]$ do
5              $sim^{\approx}(T_j, Q^t) \leftarrow sim^{\approx}(T_j, Q^t) + \frac{1}{|H|}$;
6          for $T_j \in index_i[min(h_i(Q^{t-1}))]$ do
7              $sim^{\approx}(T_j, Q^t) \leftarrow sim^{\approx}(T_j, Q^t) - \frac{1}{|H|}$;
8  foreach $j \in \{1, \ldots, |\mathcal{R}|\}$ do
9      if $T_j \notin topk^t$ and $sim^{\approx}(T_j, Q^t) > score(topk^t[k])$ then
10         update $topk^t$;
```

The time complexity of this algorithm is $O(|H| \cdot |\mathcal{R}|)$, where $|\mathcal{R}|$ is the number of objects and $|H|$ is the number of hash functions. It is achieved when the MinHash values of consecutive queries differ under all the hash functions. However, this case rarely occurs because the Jaccard similarity between consecutive queries is in fact very high.

6 Experimental results

In this section, we discuss our experimental results on a series of synthetic data sets and two real data sets. All experiments were conducted on a Mac Pro (Late 2013) server with Intel Xeon 3.70GHz CPU, 64GB memory, and 256GB SSD hard drive installed. All the algorithms were implemented in Python 3 using a much faster just-in-time compiler PyPy 3.

6.1 Results on synthetic data sets

To evaluate the effectiveness and efficiency of our two methods, the general pruning-based method ("GP" for short) and the MinHash-based method ("MHI" for short), in this section, we first report the experimental results on synthetic data sets.

We used the IBM Quest data generator[1] to produce synthetic data sets. We conducted experiments to test the efficiency and accuracy of our methods with respect to the following parameters.

- k: top-k;
- $|Q^t|$: the parameter controlling both the number of items per query and average number of items per object.
- $|\Psi|$: alphabet size;
- $|H|$: the number of different hash functions.
- $|\mathcal{R}|$: number of objects;

To generate synthetic data sets using the IBM Quest data generator, we set parameters $|Q^t|$, $|\Psi|$, and $|\mathcal{R}|$. The synthetic query stream was generated by concatenating random objects whose lengths are between $0.8|\bar{Q}^t|$ and $1.2|\bar{Q}^t|$, where $|\bar{Q}^t|$ is the average object length of the data set.

1 http://www.cs.loyola.edu/~cgiannel/assoc_gen.html.

Table 6 Values of controlled variables for tests on Jaccard similarity

Data set	Varying	k	$\lvert Q^t\rvert$	$\lvert\Psi\rvert$	$\lvert H\rvert$	$\lvert\mathcal{R}\rvert$	Figures
Synthetic	k	Multiple	10	10,000	100	100,000	3a, 4a, 6a
Synthetic	k	Multiple	10	20	100	100,000	3b, 4b, 6b
Synthetic	$\lvert Q^t\rvert$	10	Multiple	10,000	100	100,000	3d, 4d, 6c
Synthetic	$\lvert\Psi\rvert$	10	10	Multiple	100	100,000	3c, 4c, 6c
Synthetic	$\lvert H\rvert$	10	10	10,000	Multiple	100,000	3e, 6e
Synthetic	$\lvert\mathcal{R}\rvert$	10	10	10,000	100	Multiple	3f, 4e, 5a, 6f
Market basket	k	Multiple	10	16,470	100	88,162	7a–c
Market basket	$\lvert H\rvert$	10	10	16,470	Multiple	88,162	7g, h
Market basket	$\lvert\mathcal{R}\rvert$	10	10	16,470	100	Multiple	7d–f, 5b
Click stream	k	Multiple	5	17	50	31,790	8a–c
Click stream	$\lvert H\rvert$	10	5	17	Multiple	31,790	8g, h
Click stream	$\lvert\mathcal{R}\rvert$	10	5	17	50	Multiple	5c, 8d–f

The way we produce synthetic query streams mimics some real application scenarios. Take the online RPG gaming scenario as an example. Suppose one is exploring the game map and walks from one game scene to another. While each scene has its combination of enemies (as objects here), when moving from one scene to another, the most suitable query is the combination of enemies from both scenes (concatenation of the two objects within sliding window).

In each experiment, we varied one factor and fixed the others. Table 6 shows the values of controlled variables along with the corresponding figures for each test. We compare the performance of GP and MHI on the average querying time with two baseline methods, namely BFM and MHB. BFM is a brute-force method that computes the exact similarity score for every object with respect to a query. MHB is a method based on MinHash technique but does not use any indexing structures. To better illustrate how our upper bounds derived in Sect. 4 can be used to prune unpromising objects, we define the pruning effectiveness with respect to an update as follows.

Definition 3 (*Pruning effectiveness of GP*) Suppose we have $|\mathcal{R}|$ objects in total and we compute the exact similarity scores for $|\mathcal{R}|^*$ objects during an update. The pruning effectiveness with respect to this update is $1 - \frac{|\mathcal{R}|^*}{|\mathcal{R}|}$.

In this section, we report the average pruning effectiveness of 1000 updates. Moreover, we provide peak memory usage for all the methods when testing the scalability.

The accuracy in our context is defined as follows.

Definition 4 (*Accuracy*) For a method A that returns a top-k list $topk\_A^t$ with respect to query Q^t, the accuracy of A is the proportion of objects in $topk\_A^t$ whose exact similarity scores with respect to Q^t is no smaller than the similarity between $topk^t[k]$ and Q^t, where $topk^t$ is the ground truth top-k list. That is,

$$acc\_A = \frac{|\{T|T \in topk\_A^t \wedge sim(T, Q^t) \geq sim(topk^t[k], Q^t)\}|}{k}.$$

The idea here is to check the percentage of objects reported in the answer indeed have a similarity score passing the minimum similarity threshold set by the kth most similar object in the ground truth. The higher this percentage, the more accurate the answer.

The pruning algorithm GP always returns the exact query results and thus has 100 % accuracy. Since the MinHash-based methods compute similarity scores approximately, MHB and MHI give estimated answers to top-k queries. We report the average accuracy of MinHash methods in Fig. 6.

6.1.1 *Efficiency*

The average querying time of the four methods when k varies is shown in Fig. 3a, b on two synthetic data sets with a large alphabet (10^4) and a small one (20), respectively. In both cases, our baseline methods, BFM and MHB, almost have no change in average processing time. MHI that uses 100 hash functions outperforms the other four methods greatly. The pruning effectiveness of GP is shown in Fig. 4a, b when the alphabet size is set to 10^4 and 20, respectively. The pruning effectiveness of GP drops gradually because the similarity score of the kth best object in the top-k result tends to decrease when k increases. Given other parameters fixed, the similarity score of the kth best object in the result with respect to queries becomes smaller. Thus, the pruning effectiveness generally is weaker in the case of larger alphabet size.

Figure 3c provides a better illustration of the trends of the pruning effectiveness of GP when the alphabet size changes. When the alphabet size increases dramatically, the pruning effectiveness of GP drops from 69.2 to 56.3 % gradually. Thus, the average processing time of GP slightly increases. The average running time of BFM remains stable with respect to alphabet size. The average running time of MHI first decreases and then increases. When the alphabet size is small, the length of each inverted list is relatively large. Thus, a change in MinHash value of the query may result in many updates in the estimated similarity scores. When alphabet size becomes very large, though the size of each inverted list is small, the number of unmatched MinHash values between the MinHash lists of the previous query and the new query increases, which results in more searches on the inverted indices.

We also examined how the average query answering time changes with respect to average object length. The results on efficiency and pruning effectiveness are shown in Figs. 3d and 4d, respectively. The average processing time of BFM increases linearly while the performance of MHB does not increase greatly. It is because we transformed the objects of varied length into MinHash signatures of the same length. MHI still performs much better than the others and increases linearly. It is because when the average object length becomes larger, the size of each inverted list becomes larger, too, which results in longer processing time. When the average object length increases, the pruning effectiveness of GP increases dramatically, from 61.1 to 94.47 %. However, its average running time still increases slightly, since there is an increase in the cost of computing exact similarity scores for larger sets in the verification phase.

The performance with respect to the number of hash functions is shown in Fig. 3e. The average processing time of both MHB and MHI increases. MHI is roughly linear. The scalability of our methods is shown in Fig. 3f. The runtime of all the methods increases almost linearly as the number of object increases. The pruning effectiveness of GP increases from 48.3 to 64.2 % when the number of objects increases from 10^4 to 10^6, which is shown in Fig. 4e.

The peak memory usage of our methods is shown in Fig. 5a. BFM and GP follow the similar growth trend and GP requires only a little more memory than BFM. It is because we

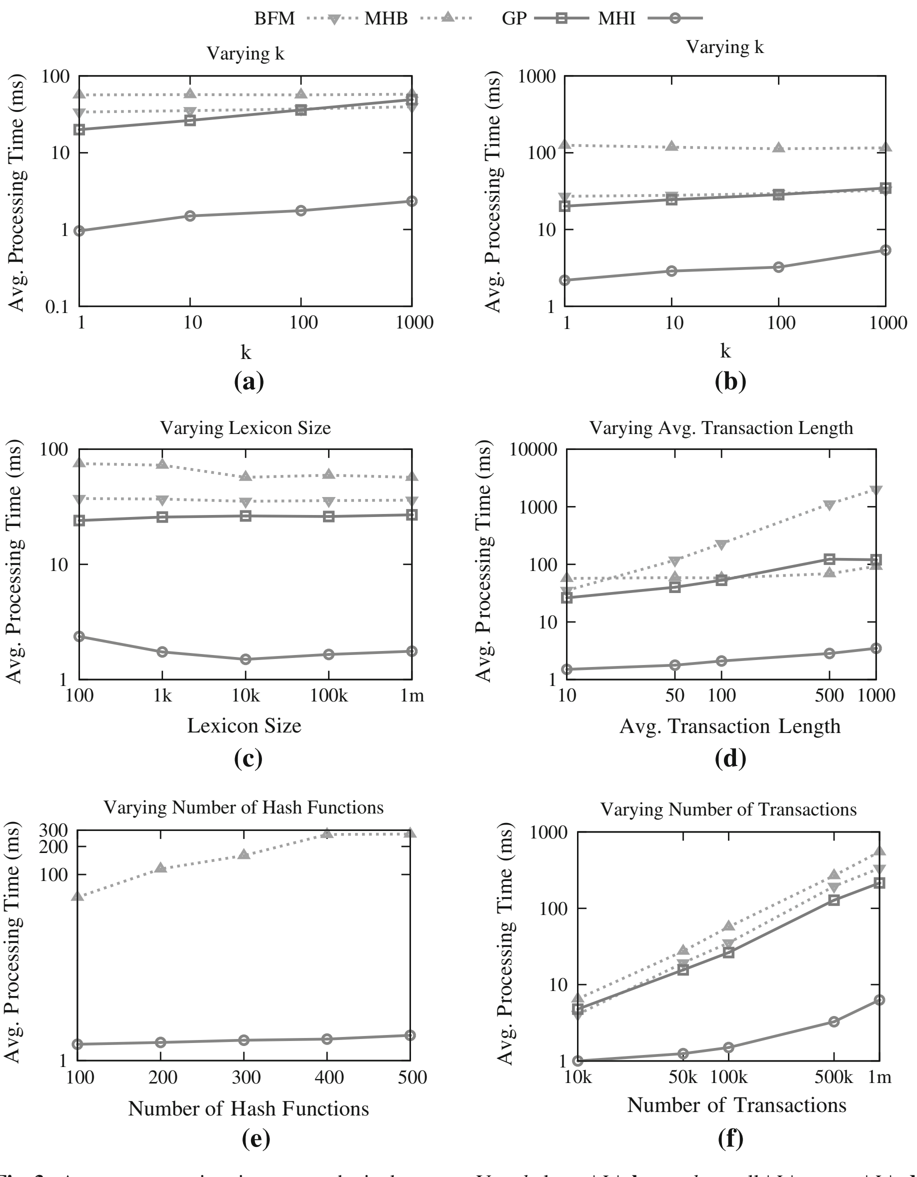


Fig. 3 Average processing time on synthetic data set. **a** Vary k, large $|\Psi|$, **b** vary k, small $|\Psi|$, **c** vary $|\Psi|$, **d** vary $|Q^t|$, **e** vary $|H|$, **f** vary $|\mathcal{R}|$

use extra data structures like queue and max-heap to store the upper bounds of similarity scores. Compared to BFM and GP, MinHash-based methods need more space since we need to store a MinHash signature for each object. MHB costs less space than MHI because we store MinHash signatures differently for these two methods. For MHB, we simply store each MinHash signature as an array, while in MHI, we construct an inverted index for each hash function to enable more efficient update, which requires more space.

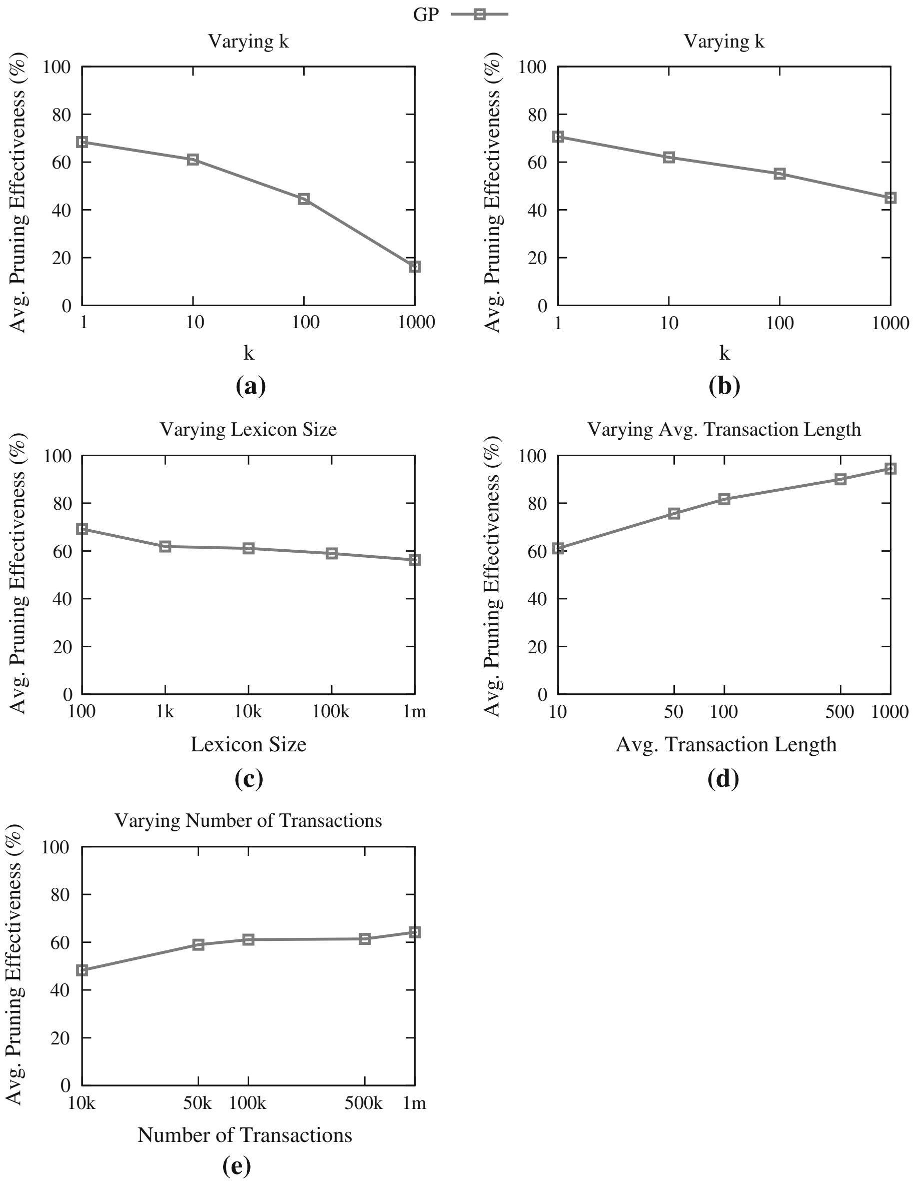


Fig. 4 Pruning effectiveness of GP on synthetic data set. **a** Vary k, large $|\Psi|$, **b** vary k, small $|\Psi|$, **c** vary $|\Psi|$, **d** vary $|H|$, **e** vary $|\mathcal{R}|$

6.1.2 Accuracy

We also tested the accuracy of the MinHash-based algorithms. By our definition, the accuracy of the two MinHash-based algorithms are the same according to the same set of hash functions. We denote the MinHash methods by MH in the figures that report accuracy. The accuracy can be affected by two factors, the number of hash functions used and the intrinsic characteristics of our data set, such as the distribution of similarity scores among objects.

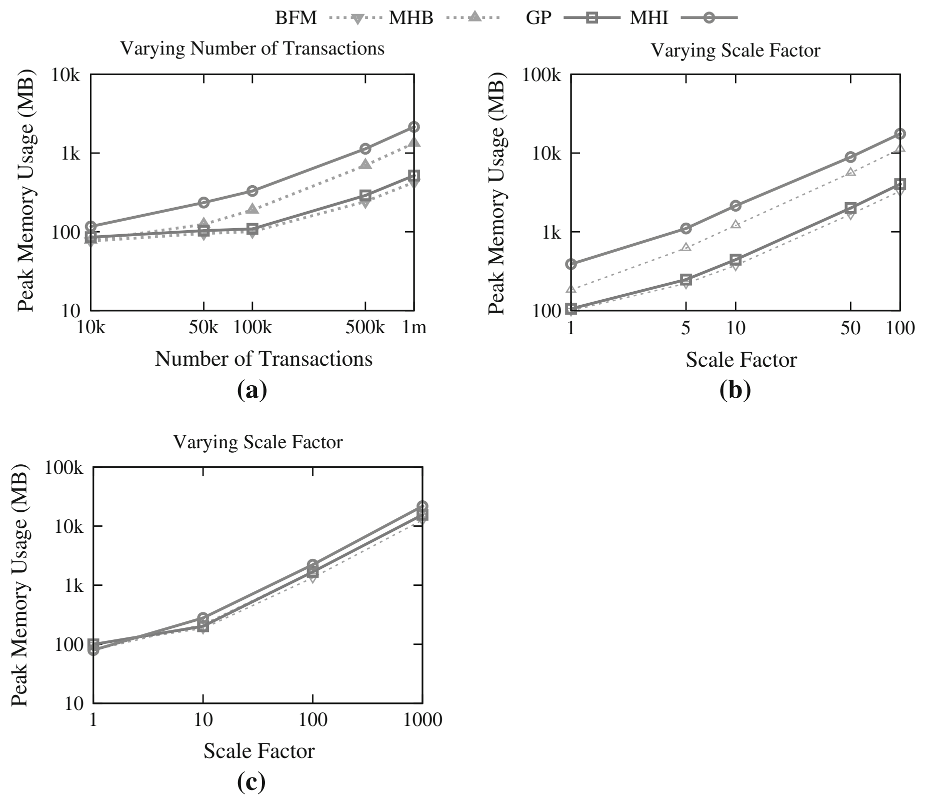


Fig. 5 Data set peak memory usage on different data sets. **a** Synthetic data set, **b** market basket data set, **c** click stream

The results show that the MinHash-based methods achieve an accuracy ranging from 68 to 98 % on the synthetic data sets when 100 hash functions are used. Figure 6a, b shows the change in accuracy when *k* increases on data sets of large alphabet and small alphabet, respectively. On the data set of alphabet size 10^4, the accuracy decreases gradually, from 98 to 68.7 %. When alphabet size is 20, the average accuracy does not vary much but still in a slightly descending trend.

The average accuracy increases quickly and then decreases when the alphabet size increases, as shown in Fig. 6c. The highest accuracy is achieved when the alphabet size is 10^4. The reason is as follows. With the fixed number of objects, when alphabet size is small, there are many similar objects in the data set, which lowers the accuracy. In contrast, when the alphabet size is too large, objects in the data set would be less similar and lead to slightly lower accuracy. In Fig. 6c, when the average object length increases from 10 to 10^3, the accuracy drops drastically from 86.7 to 10 %. The reason is apparent. In MinHash, the Jaccard similarity is the probability of two objects having the same minimal hash values with respect to all the hash functions. With a fixed number of hash functions, the larger the window size (average object length here), the less the portion of the elements in the window are hashed as the common MinHash value, which leads to a less accurate estimation. Thus, more hash functions are needed to achieve the same level of accuracy when the object length becomes larger.

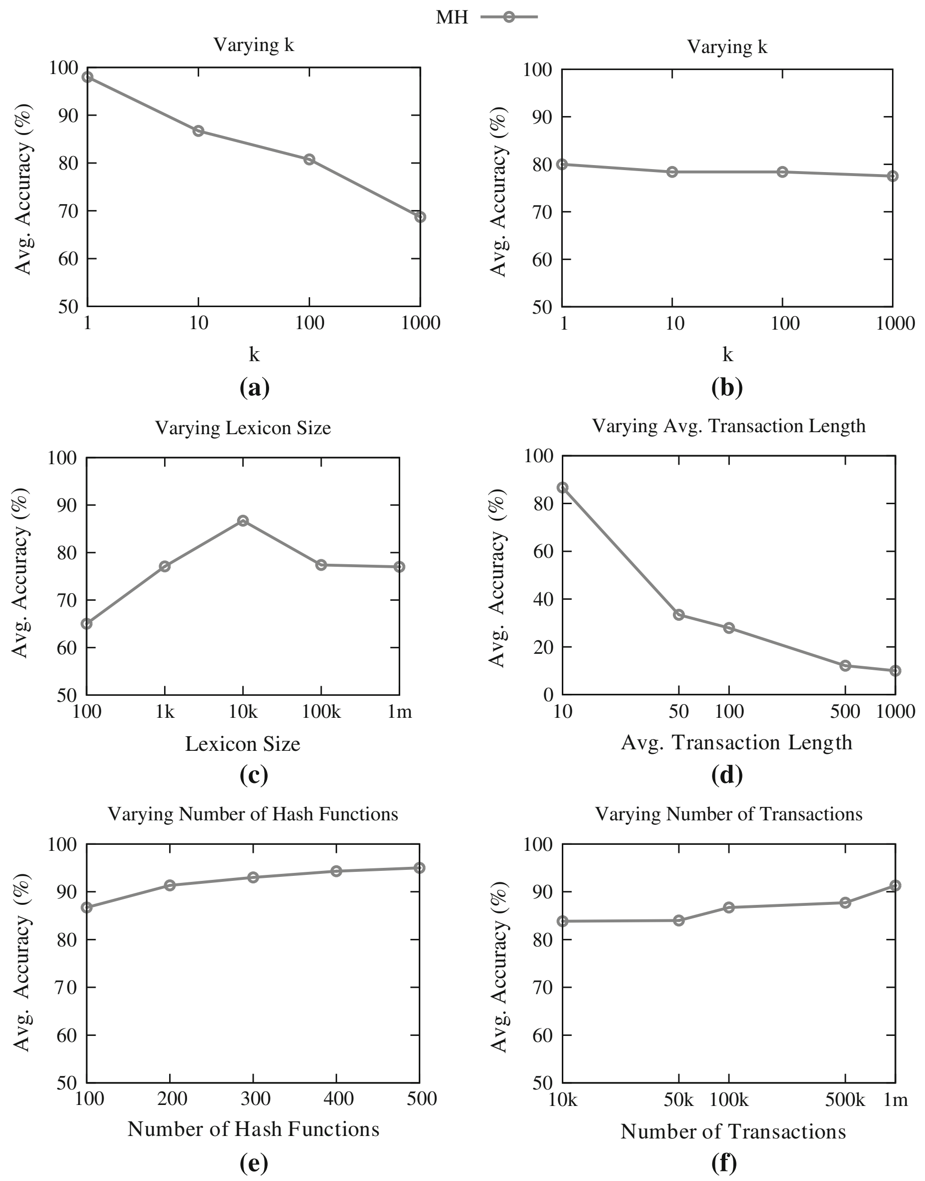


Fig. 6 Average accuracy of hashing-based method on synthetic data set. **a** Vary k, large $|\Psi|$, **b** vary k, small $|\Psi|$, **c** vary $|\Psi|$, **d** vary $|Q^t|$, **e** vary $|H|$, **f** vary $|\mathcal{R}|$

We also tested the trend of accuracy when the number of hash functions and the number of objects increase. The results are shown in Fig. 6e and f, respectively. When more hash functions are used, the estimated Jaccard similarity is closer to the exact score and thus leads to a higher accuracy. As shown in Fig. 6e, the result follows this trend. Our methods achieve average accuracy rate ranging from 86.7 to 95 %. Figure 6f suggests that the average accuracy increases gradually when the number of objects increases.

Table 7 Statistics of the real data sets

Data set	$\vert\mathcal{R}\vert$	Avg. $\vert T_i\vert$	$\vert\Psi\vert$
Market basket	88,162	10.306	16, 470
Click stream	31,790	5.3338	17

6.2 Results on real data sets

We conducted experimental studies on two real data sets: a Market Basket data set[2] from an anonymous Belgian retail store, and a Click Stream data set[3] where the predefined category, such as "news" and "tech," replaces each URL on the MSNBC website. The data processing was conducted by the provider. Moreover, the provider of the Click Stream data set also removed those short sequences of lengths no larger than 8. We then transformed each sequence into a set. For Click Stream data set, each static object is a set of categories for URLs visited by each user, while for the market basket data set, each static object is a set of items purchased by a customer. There is a significant difference in alphabet size of the two real data sets. The market basket data set has 16,470 distinct items, while the Click Stream data set contains only 17 distinct items. Table 7 shows the detailed statistics of the data sets.

To generate the querying stream, we used the same method when generating querying streams for synthetic data sets. That is, we concatenated objects randomly selected in the data set whose size is between $0.8|\bar{Q}^t|$ and $1.2|\bar{Q}^t|$, where $|\bar{Q}^t|$ is the average object length of the data set. For each data set, we compared the average processing time and the average accuracy when k or $|H|$ varies, respectively. We also include scalability test on both time and peak memory usage. The results are shown in Figs. 5, 7, and 8.

The trends of the curves for the market basket data set are consistent with the results on the synthetic data sets. To compare the average processing time of different methods when k varies, we set the default number of hash functions to 100. The processing time of BFM and MHB is stable. MHI always has the best performance, and the processing time increases gradually. The pruning effectiveness of GP decreases from 64.9 to 38.3 % gradually when k increases. Therefore, the processing time of GP increases when k increases. The MinHash-based algorithms can always achieve accuracy over 70 % when k varies from 1 to 1000, which is shown in Fig. 7b. Figure 7g, h shows the trends of processing time and accuracy of the MinHash-based methods, respectively, when different numbers of hash functions are used. The average processing time of both MHB and MHI increases almost linearly with the number of hash functions. The performance of MHI is between 31 and 96 times faster than MHB. The average accuracy increases steadily from 73.1 to 93.4 % when the number of hash functions increases from 100 to 500.

We also tested the scalability on this data set by duplicating the data set 5, 10, 50, and 100 times. We report the average processing time, average accuracy, average pruning effectiveness, and peak memory usage in Figs. 7d–f and 5b, respectively. Figure 7d shows that the processing time of all the methods increases linearly as the data set size increases. The accuracy first increases drastically and then reaches the level of over 95 % and remains steady. The pruning effectiveness does not change much and is around 60 %.

[2] http://fimi.ua.ac.be/data/.

[3] http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php.

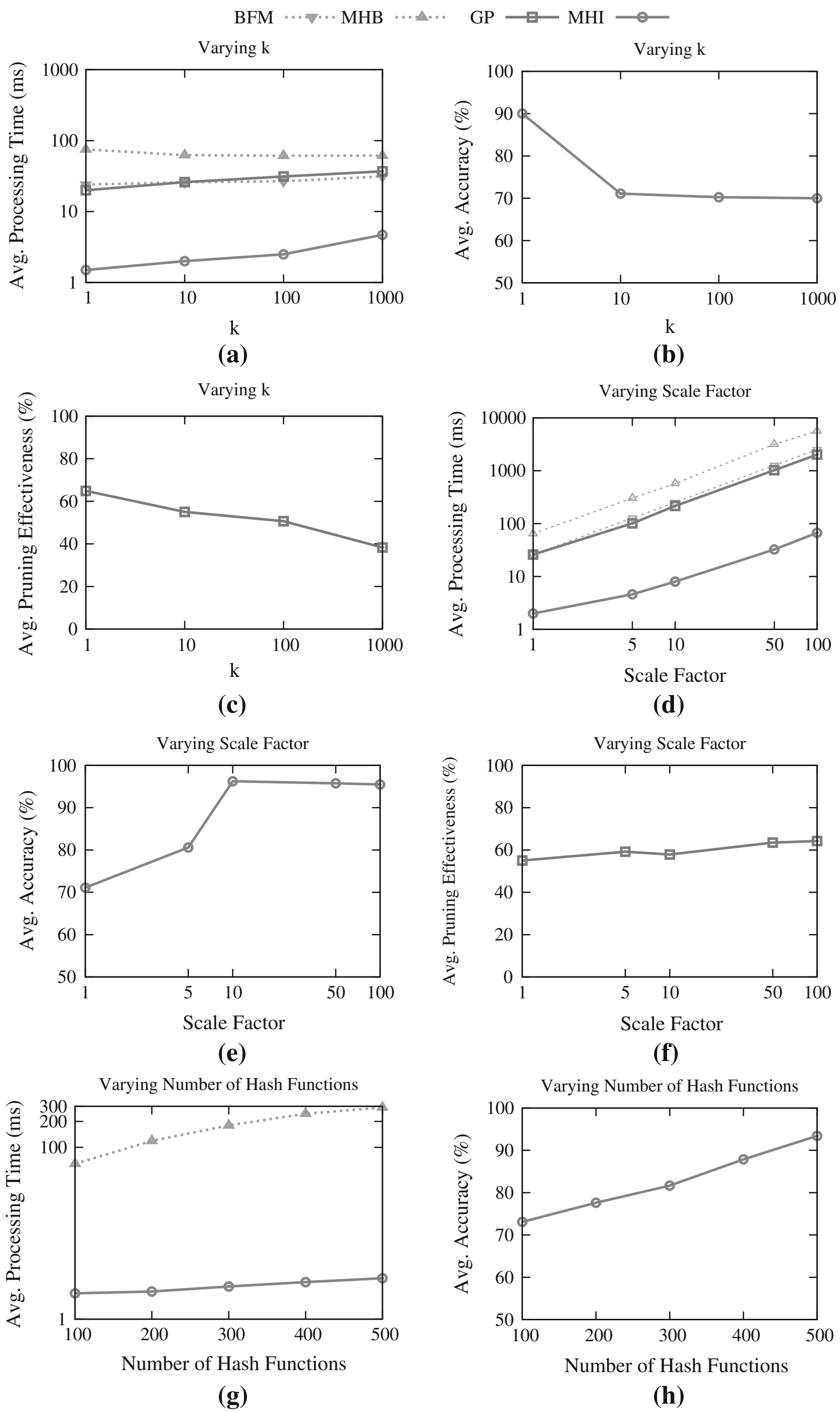


Fig. 7 Results on market basket data set. **a** Vary k (time), **b** vary k (accuracy), **c** vary k (pruning effectiveness), **d** vary $|\mathcal{R}|$ (time), **e** vary $|\mathcal{R}|$ (accuracy), **f** vary $|\mathcal{R}|$ (pruning effectiveness), **g** vary $|H|$ (time), **h** vary $|H|$ (accuracy)

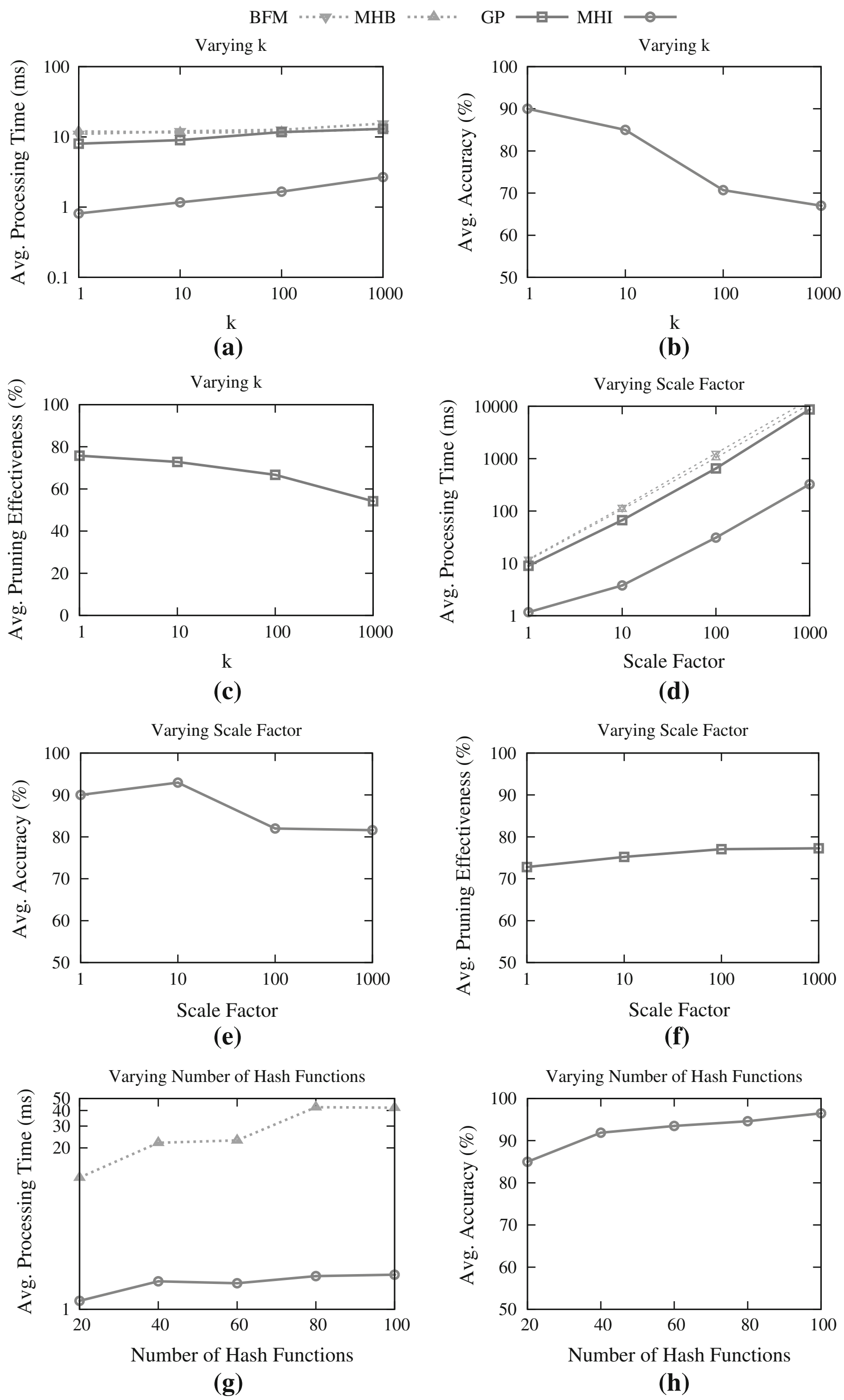


Fig. 8 Results on click stream data set. **a** Vary k (time), **b** vary k (accuracy), **c** vary k (pruning effectiveness), **d** vary $|\mathcal{R}|$ (time), **e** vary $|\mathcal{R}|$ (accuracy), **f** vary $|\mathcal{R}|$ (pruning effectiveness), **g** vary $|H|$ (time), **h** vary $|H|$ (accuracy)

The peak memory usage is shown in Fig. 5b. The peak memory size used of all the methods increases almost linearly when the data set increases. Similar to the results in the scalability test on the synthetic data sets, GP uses only a little more memory than BFM. Their trends are similar. The MinHash-based methods consume more space than BFM and GP. The reason is the same as explained on the synthetic data sets in Sect. 6.1.1.

The results on the click stream data set are shown in Figs. 8 and 5c. Since the average length of the objects in this data set is only 5.33, the MinHash-based algorithms do not need many hash functions in order to achieve high accuracy. As shown in Fig. 8h, the average accuracy is higher than 90 % when 40 or more hash functions are used. To test the performance when k varies, we used 50 hash functions as the default. When k varies, MHI is still the best method with respect to average query answering time. The average accuracy drops gradually from 90 to 67 %. As shown in Fig. 8c, the pruning effectiveness of GP decreases from 75.8 to 54.2 %, which is generally better than the results on the market basket data set.

We tested the scalability on the click stream data set by duplicating the data set 10, 100, and 1000 times. Figures 8d–f and 5c show how the average processing time, average accuracy, average pruning effectiveness, and peak memory usage change with respect to different scale of data set, respectively. The average processing time of all the methods increases almost linearly when the data set size increases. MHI always achieves the lowest processing time and outperforms the baselines by more than an order of magnitude. Moreover, it reports with reasonably high accuracy in the range of 81.6 and 90 % as shown in Fig. 8e. When the data set size increases, the pruning effectiveness of GP increases steadily and is generally higher than the results on the market basket data set.

The memory usage of BFM, GP, and MHB is very close and the corresponding curves in Fig. 5c overlap with each other. Comparing to the other data sets, the memory usage of MHB is much closer to BFM and GP because we use less number of hash functions. MHI still consumes more memory due to the inverted indices data structure. However, its memory usage is much closer to the other three methods comparing to the results on the other data sets. Recall that the size of our indexing structure depends on the number of hash functions used and the alphabet size.

6.3 Summary

We conclude the following from our experiments. First, the pruning-based method is an exact method and the hashing-based method can achieve very good accuracy when a few hundreds of hash functions are used. The hashing-based method can lead to good accuracy and can achieve faster average querying time than the pruning-based method in most cases. However, when k increases, the accuracy drops and the running time increases greatly. The pruning-based method runs slower but maintains relatively stable running time with increasing k. For the hashing-based method, usually larger number of hash functions $|H|$ provides better accuracy but also takes more running time. However, when the number of hash functions exceeds 1,000, the increase in accuracy usually is very minor. As shown on synthetic data set, when the average object length $|Q^t|$ becomes larger, the accuracy of the hashing-based method drops greatly while the pruning effectiveness of the pruning-based method increases significantly, which implies that the pruning-based method is more feasible in this case. In general, both methods achieve better performance on data sets with larger lexicon size $|\Psi|$. The sparser the data, the more significant the difference on similarity, and usually the better runtime performance.

7 Conclusions

In this paper, we studied the problem of continuous similarity search for evolving queries. To the best of our knowledge, it is the first research endeavor on this problem. We devised two efficient methods with different framework. The pruning-based method uses pruning strategies to reduce the cost of computing the exact similarity scores. The MinHash-based method approximates the Jaccard similarity score based on MinHash technique and efficiently updates the estimated scores using indexing structures. The empirical results on synthetic and real data sets verify the effectiveness and efficiency of our methods.

As for future work, we plan to consider the following interesting directions.

- *Enhance the pruning effectiveness* For example, we may incorporate the evolving feature into the pruning techniques used in static similarity join and search algorithms to further prune candidates.
- *Improve the hashing-based method for other similarity measures* We may extend our hashing-based framework to other similarity and distance measures that can be approximated by an LSH scheme used in MinHash.
- *Similarity search over multiple evolving streams* Given a static object and multiple data streams with a fixed sliding window size, we may want to continuously find the top-k data streams whose last n items are most similar to the static object. The techniques proposed in this paper can be further extended for the problem.

Appendix: Other similarity measures and their upper bounds for pruning method

In this section, we extend the upper bounds for pruning method to weighted overlap, weighted cosine, and weighted dice similarity. Similar to the case of weighted Jaccard similarity, they all have monotonicity with respect to number of updates u.

Property 1 (A progressive upper bound for weighted overlap similarity). *We first define the weighted overlap similarity as follows.*

$$sim_{over}(X, Y) = sim_{over}(\vec{X}, \vec{Y}) = \sum_{i=1}^{|\Psi|} \min(x_i, y_i) \tag{8}$$

Let X, Y be two multisets and Y' be the multiset with u updates on Y. Given $|X|$, $|Y|$, and the weighted overlap similarity score $sim_{over}(X, Y)$ between X and Y, without the knowledge of the updated elements in Y', we have an upper bound for $sim_{over}(X, Y')$.

$$sim_{over}(X, Y') \leq sim_{over}(X, Y) + u \tag{9}$$

Proof By definition,

$$sim_{over}(X, Y) = \sum_{i=1}^{|\Psi|} \min(x_i, y_i)$$

Obviously, the maximum possible increase after u updates is u. □

Property 2 (A progressive upper bound for weighted cosine similarity) *We first define the weighted cosine similarity as follows.*

$$sim_{cos}(X, Y) = sim_{cos}(\vec{X}, \vec{Y}) = \frac{\sum_{i=1}^{|\Psi|} \min(x_i, y_i)}{\sqrt{|X||Y|}} \tag{10}$$

Let X, Y be two multisets and Y' be the multiset with u updates on Y. Given $|X|$, $|Y|$, and the weighted cosine similarity score $sim_{cos}(X, Y)$ between X and Y, without the knowledge of the updated elements in Y', we have an upper bound for $sim_{cos}(X, Y')$.

$$sim_{cos}(X, Y') \leq sim_{cos}(X, Y) + \frac{u}{\sqrt{|X||Y|}} \tag{11}$$

Proof In our scenario, $|Y| = |Y'|$. Thus, $\sqrt{|X||Y|} = \sqrt{|X||Y'|}$. By Property 1, we have

$$sim_{cos}(X, Y') \leq \frac{\sum_{i=1}^{|\Psi|} \min(x_i, y_i) + u}{\sqrt{|X||Y'|}} = sim_{cos}(X, Y) + \frac{u}{\sqrt{|X||Y|}}$$

□

Property 3 (A progressive upper bound for weighted dice similarity) *We first define the weighted dice similarity as follows.*

$$sim_{dice}(X, Y) = sim_{dice}(\vec{X}, \vec{Y}) = \frac{2\sum_{i=1}^{|\Psi|} \min(x_i, y_i)}{|X| + |Y|} \tag{12}$$

Let X, Y be two multisets and Y' be the multiset with u updates on Y. Given $|X|$, $|Y|$, and the weighted dice similarity score $sim_{dice}(X, Y)$ between X and Y, without the knowledge of the updated elements in Y', we have an upper bound for $sim_{dice}(X, Y')$.

$$sim_{dice}(X, Y') \leq sim_{dice}(X, Y) + \frac{2u}{|X| + |Y|} \tag{13}$$

Proof In our scenario, $|Y| = |Y'|$. Thus, $|X| + |Y| = |X| + |Y'|$. By Property 1, we have

$$sim_{dice}(X, Y') \leq \frac{2\sum_{i=1}^{|\Psi|} \min(x_i, y_i) + 2u}{|X| + |Y'|} = sim_{dice}(X, Y) + \frac{2u}{|X| + |Y|}$$

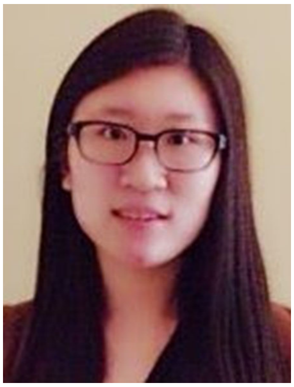
□

References

1. Andoni A, Indyk P (2006) Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In: Proceedings of the 47th annual IEEE symposium on foundations of computer science, FOCS '06, Washington, DC, USA. IEEE Computer Society, pp 459–468
2. Artin E (2011) Geometric algebra. Wiley, Hoboken
3. Bayardo RJ, Ma Y, Srikant R (2007) Scaling up all pairs similarity search. In: Proceedings of the 16th international conference on World Wide Web, WWW '07, New York, NY, USA. ACM, pp 131–140
4. Böhm C, Ooi BC, Plant C, Yan Y (2007) Efficiently processing continuous k-nn queries on data streams. In: Proceedings of the international conference on data engineering, ICDE '07, Washington, DC, USA. IEEE Computer Society, pp 156–165
5. Broder A (1997) On the resemblance and containment of documents. In: Proceedings of the compression and complexity of sequences 1997, SEQUENCES '97, Washington, DC, USA. IEEE Computer Society, pp 21–29
6. Broder AZ, Charikar M, Frieze AM, Mitzenmacher M (2000) Min-wise independent permutations. J Comput Syst Sci 60(3):630–659
7. Charikar MS (2002) Similarity estimation techniques from rounding algorithms. In: Proceedings of the thirty-fourth annual ACM symposium on theory of computing, STOC '02, New York, NY, USA. ACM, pp 380–388

8. Chaudhuri S, Ganti V, Kaushik R (2006) A primitive operator for similarity joins in data cleaning. In: Proceedings of the 22nd international conference on data engineering, ICDE '06, Washington, DC, USA. IEEE Computer Society, pp 5–15
9. Cohen WW (1998) Integration of heterogeneous databases without common domains using queries based on textual similarity. In: Proceedings of the 1998 ACM SIGMOD international conference on management of data, SIGMOD '98, New York, NY, USA. ACM, pp 201–212
10. Cost S, Salzberg S (1993) A weighted nearest neighbor algorithm for learning with symbolic features. Mach Learn 10(1):57–78
11. Datar M, Muthukrishnan S (2002) Estimating rarity and similarity over data stream windows. In: Proceedings of the 10th annual European symposium on algorithms, ESA '02. Springer, London, pp 323–334
12. Devroye L, Wagner TJ (1982) 8 nearest neighbor methods in discrimination. Handbook of statistics 2:193–197
13. Faloutsos C, Barber R, Flickner M, Hafner J, Niblack W, Petkovic D, Equitz W (1994) Efficient and effective querying by image content. J Intell Inf Syst 3(3–4):231–262
14. Faloutsos C, Oard DW (1995) A survey of information retrieval and filtering methods. University of Maryland at College Park, College Park, MD, USA. Univ. of Maryland Institute for Advanced Computer Studies Report
15. Flickner M, Sawhney H, Niblack W, Ashley J, Huang Q, Dom B, Gorkani M, Hafner J, Lee D, Petkovic D, Steele D, Yanker P (1995) Query by image and video content: the qbic system. Computer 28(9):23–32
16. Gersho A, Gray RM (1991) Vector quantization and signal compression. Kluwer Academic Publishers, Norwell
17. Gionis A, Indyk P, Motwani R (1999) Similarity search in high dimensions via hashing. In: Proceedings of the 25th international conference on very large data bases, VLDB '99, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc., pp 518–529
18. Hastie T, Tibshirani R (1995) Discriminant adaptive nearest neighbor classification. In: Proceedings of the first international conference on knowledge discovery and data mining, KDD '95, Palo Alto, CA, USA. AAAI Press, pp 142–149
19. Henzinger M (2006) Finding near-duplicate web pages: a large-scale evaluation of algorithms. In: Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval, SIGIR '06, New York, NY, USA. ACM, pp 284–291
20. Indyk P (2001) A small approximately min-wise independent family of hash functions. J Algorithms 38(1):84–90
21. Indyk P, Motwani R (1998) Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the thirtieth annual ACM symposium on theory of computing, STOC '98, New York, NY, USA. ACM, pp 604–613
22. Koivune V, Kassam S (1995) Nearest neighbor filters for multivariate data. In: IEEE workshop on nonlinear signal and image processing, Washington, DC, USA. IEEE Computer Society, pp 734–737
23. Kollios G, Tsotras VJ (2002) Hashing methods for temporal data. IEEE Trans Knowl Data Eng 14(4):902–919
24. Kontaki M, Papadopoulos AN (2004) Efficient similarity search in streaming time sequences. In: Proceedings of the 16th international conference on scientific and statistical database management, SSDBM '04, Washington, DC, USA. IEEE Computer Society, pp 63–72
25. Koudas N, Ooi BC, Tan K-L, Zhang R (2004) Approximate nn queries on streams with guaranteed error/performance bounds. In: Proceedings of the thirtieth international conference on very large data bases, VLDB '04. VLDB Endowment, pp 804–815
26. Lian X, Chen L, Wang B (2007) Approximate similarity search over multiple stream time series. In: Proceedings of the 12th international conference on database systems for advanced applications, DASFAA'07. Springer, Berlin, pp 962–968
27. Mouratidis K, Bakiras S, Papadias D (2006) Continuous monitoring of top-k queries over sliding windows. In: Proceedings of the 2006 ACM SIGMOD international conference on management of data, SIGMOD '06, New York, NY, USA. ACM, pp 635–646
28. Mouratidis K, Papadias D (2007) Continuous nearest neighbor queries over sliding windows. IEEE Trans Knowl Data Eng 19(6):789–803
29. Mouratidis K, Papadias D, Bakiras S, Tao Y (2005) A threshold-based algorithm for continuous monitoring of k nearest neighbors. IEEE Trans Knowl Data Eng 17(11):1451–1464
30. Pan S, Zhu X (2012) Continuous top-k query for graph streams. In Proceedings of the 21st ACM international conference on information and knowledge management. CIKM '12, New York, NY, USA. ACM, pp 2659–2662
31. Pentland A, Picard RW, Sclaroff S (1994) Photobook: content-based manipulation of image databases. In: Storage and retrieval for image and video databases, Bellingham, WA, USA. SPIE, pp 34–47

32. Rao W, Chen L, Chen S, Tarkoma S (2014) Evaluating continuous top-k queries over document streams. World Wide Web 17(1):59–83
33. Salton G, McGill MJ (1986) Introduction to modern information retrieval. McGraw-Hill Inc., New York
34. Sarawagi S, Kirpal A (2004) Efficient set joins on similarity predicates. In: Proceedings of the 2004 ACM SIGMOD international conference on management of data, SIGMOD '04, New York, NY, USA. ACM, pp 743–754
35. Smeulders A, Jain R (eds) (1998) Image databases and multimedia search. World Scientific Publishing Co., Inc., River Edge
36. Sun Y, Han J, Yan X, Yu PS, Wu T (2011) Pathsim: meta path-based top-k similarity search in heterogeneous information networks. Proc VLDB Endow 4(11):992–1003
37. Winkler WE (1999) The state of record linkage and current research problems. Statistical Research Division, US Census Bureau, Suitland
38. Xiao C, Wang W, Lin X, Yu JX (2008) Efficient similarity joins for near duplicate detection. In: Proceedings of the 17th international conference on World Wide Web, WWW '08, New York, NY, USA. ACM, pp 131–140



Xiaoning Xu received her Master's degree in computing science from Simon Fraser University, Canada in 2014, supervised by Prof. Jian Pei. Previously, she was in the SFU-ZJU computing science dual degree program. She received a B.Sc. degree from Simon Fraser University, Canada, and a B.Eng. degree from Zhejiang University, China, both in 2012. Her research interests focus on similarity search and data stream processing. She is currently a software engineer at Fortinet Inc., working on data analytics and threat detection in network log data.

Chuancong Gao is a Ph.D. student in computing science in Simon Fraser University, Canada, since 2013, supervised by Prof. Jian Pei. Previously he received a master degree from Tsinghua University, China, in 2010 and a bachelor degree from Shandong University, China, in 2007, both in computer science and in engineering. His current research interests are data cleaning and data integration.

Jian Pei is the Canada Research Chair in Big Data Science and a professor at the School of Computing Science and the Department of Statistics and Actuarial Science at Simon Fraser University, Canada. He received a Ph.D. degree at the same school in 2002 under Dr. Jiawei Han's supervision. His research interests are to develop effective and efficient data analysis techniques for novel data intensive applications. He has published prolifically and is one of the top cited authors in data mining. He received a series of prestigious awards. He is also active in providing consulting service to industry and transferring the research outcome in his group to industry and applications. He is an editor of several esteemed journals in his areas and a passionate organizer of the premier academic conferences and initiatives defining the frontiers of the areas. He is an IEEE fellow.

Ke Wang received Ph.D. from Georgia Institute of Technology. He is currently a professor at the School of Computing Science, Simon Fraser University. His research interests include database technology, data mining, and knowledge discovery, with emphasis on massive data sets, graph and network data, and data privacy. Ke Wang has published more than 100 research papers in database, information retrieval, and data mining conferences. He is currently an associate editor of the ACM TKDD journal.

Abdullah Al-Barakati was the head of Information Systems Department at King Abdulaziz University (KAU), Jeddah, Saudi Arabia, since in the period 2014–2015. He is also a faculty member and a lecturer in the Faculty of Computing and Information Technology since the year 2013. Dr. Al-Barakati's academic interests range from teaching and academic research to modern theories in applied informatics. His research interests revolve on the area of Big Data ranging from modern practices in Big Data to advanced concepts and special issues in Big Data mining (pattern extraction, visualization, and analysis). Language Processing (LP) and Machine Learning (ML) also fall within his research interests as he has developed a passion for text analysis and manipulation. Furthermore, web applications especially the dynamics of Social Media and web 2.0 are important elements of Dr. Al-Barakati's research interests as he believes that they form the cornerstones of many astonishing innovations now and in the future.