Dynamic Set kNN Self-Join

Daichi Amagata
Osaka University
amagata.daichi@ist.osaka-u.ac.jp

Takahiro Hara
Osaka University
hara@ist.osaka-u.ac.jp

Chuan Xiao
Nagoya University
chuanx@nagoya-u.jp

***Abstract*—In many applications, data objects can be represented as sets. For example, in video on-demand and social network services, the user data consists of a set of movies that have been watched and a set of users (friends), respectively, and they can be used for recommendation and information extraction. The problem of set similarity self-join hence has been studied extensively. Existing studies assume that sets are static, but in the above applications, sets are dynamically updated, and this requires continuous updating the join result. In this paper, we study a novel problem, dynamic set kNN self-join, i.e., for each set, we continuously compute its k nearest neighbor sets. Our problem poses a challenge for the efficiency of computation, because just an element insertion (deletion) into (from) a set may affect the kNN results of many sets. To address this challenge, we first investigate the property of the dynamic set kNN self-join problem to observe the search space derived from a set update. Then, based on this observation, we propose an efficient algorithm. This algorithm employs an indexing technique that enables incremental similarity computation and prunes unnecessary similarity computation. Our empirical studies using real datasets show the efficiency and scalability of our algorithm.**

I. Introduction

Given a collection of sets, each of which consists of elements, the problem of set similarity self-join retrieves all similar pairs of two sets in the collection, and two sets are similar if their similarity satisfies a user-specified threshold. In this paper, we study a variant of the set similarity self-join problem, i.e., dynamic set k nearest neighbor (kNN) self-join problem. Given a collection of sets and k, this problem monitors the k most similar sets in the collection of sets *for each set*. In this paper, we focus on Jaccard similarity and Cosine similarity, which have commonly been considered in previous works [1], [2], [3], [4], [5], [6].

Motivation. Although existing works consider static data, recent Web-based applications usually deal with dynamic data [7], [8]. It is worth noting that one of the most important tools for Web applications is (nearest neighbor) join [9]. As illustrated in the motivating examples below, their requirement can be a technique that *continuously* computes set kNN self-join.

- *Online collaborative filtering*. Similarity join techniques are practically required because of the fact that real services, e.g., Netflix, YouTube, and Facebook, employ recommender systems that provide personalized recommendation *for all (active) users* based on collaborative filtering (CF) [10], [11], [12]. Note that, for all users, CF obtains similar preferences to those of them, and utilizes the preferences to determine recommendation items [13]. For example, in video recommendation services, sets and elements are respectively movies (or users) and users (or movies), and (the k most) similar sets are regarded as the preferences [14]. Although CF is a well-known application of set similarity join [1], [2], [5], [14], [15], [16], static CF is not suitable for streaming recommendation, which has been receiving much attention [17], [18], [19]. Because recommendation services dynamically generate data, recommender systems should take into account the latest data and provide real-time recommendation. Online CF is therefore required, and its task is to continuously update similar sets (i.e., users/items) for each set. Dynamic kNN self-join satisfies this task, and the monitoring results are used for updating preferences and making real-time recommendations.
- *Online social network analysis*. Social network analysis plays an important role in mining human communities [20] and social filtering (i.e, enhancing collaborative filtering) [21], [22]. In this setting, a set and an element respectively correspond to a user and a group that he/she belongs to, and obviously, new (some) elements are dynamically inserted (deleted) into (from) corresponding sets. This analysis is done based on a similarity join result [9], [23]. To make social network analysis effective, the up-to-date network structures have to be reflected [24], thus dynamic set kNN self-join is useful. This is because it continuously updates the join result based on the up-to-date network structures. The results are exploited for clustering similar sets to mine human communities and to recommend potential friends [25].

The requirement of the above applications is that each set has the sets most similar to it. Set similarity self-join may provide no results for some sets, because the join result is dependent on a user-specified threshold. Therefore, set similarity self-join is not appropriate for the above applications.

Challenge. To the best of our knowledge, the problem of (dynamic) set kNN self-join has not been addressed in literatures. Existing techniques for set similarity self-join are unfortunately inefficient for our problem. They assume static sets and computing join result from scratch whenever a set is updated is prohibitive. Besides, the techniques rely on pre-sorting of element frequency and set size [5], [6], [26] to obtain good performances. Since both orders are frequently updated in dynamic environments, if this technique is employed, we need many frequency counting and sorting operations, which is computationally expensive.

In addition, the difficulty of our problem derives from the fact that a new element insertion (and also an element deletion) may affect the kNN results of many sets. Assume that a set s has a new element. In this case, it is trivial that the kNN sets of s may change. Furthermore, since the similarity between s and a given set s' also varies, s may be newly included in or removed from the kNN result of s'. If s is removed from the kNN result of s', s' needs to find a new kNN result. It may be considered that, for each set, finding its kNN set with a top-k set similarity search algorithm overcomes this challenge. However, in the applications mentioned before, the number of (active) sets is large [11], [12], so this approach cannot satisfy the requirement of real-time monitoring. For example, even if a top-k set similarity search algorithm takes one millisecond to find the kNN set of a given set, it takes 1,000 seconds to obtain the join result for a collection of one million sets.

Contribution. In this paper, we overcome the above challenges and make the following contributions.

- *Fundamental property*. Assume that s has a new element e, then the kNN result of s may change. This poses two questions: which sets can newly become the kNN result of s and whose kNN results do we have to update? We answer these questions, and to efficiently access them, an inverted index and reverse kNN lists are employed (Section III).
- *LI-DSN-Join* (Local-Index-based dynamic set kNN self-join). The main bottleneck of set kNN self-join is similarity computation between sets. To remove this bottleneck and incrementally update the similarity (i.e., $O(1)$ time), we propose an algorithm that indexes, for each set s, the size of the differences between s and s' ($|s \backslash s'|$). Furthermore, we provide a cost model to determine the threshold for the index, i.e., which $|s \backslash s'|$ has to be indexed by s to efficiently update the kNN result while guaranteeing the correct answer. We theoretically validate the design of LI-DSN-Join, i.e., its efficiency (Section IV).
- *Empirical evaluation*. We conduct extensive experiments using ten real datasets (Section V). We compared LI-DSN-Join with existing techniques for exact set similarity join, ALL [14], and for exact set similarity search, Tree [6]. Our experimental results demonstrate that LI-DSN-Join updates the kNN join result faster than them in all the tests.

In addition to the above contents, we provide our problem definition in Section II, review related works in Section VI, and conclude this paper in Section VII.

II. Preliminary

Consider a finite universe of elements $\mathcal{E} = \{e_1, e_2, ..., e_{|\mathcal{E}|}\}$. A set s is a subset of $\mathcal{E}$. Given two sets s and s', $\mathsf{Sim}(\cdot,\cdot) \in [0,1]$ measures their similarity. In this paper, as $\mathsf{Sim}(\cdot,\cdot)$, we use Jaccard similarity and Cosine similarity, which are respectively defined as

$$\mathsf{Jac}(s,s') = \frac{|s \cap s'|}{|s \cup s'|} \text{ and } \mathsf{Cos}(s,s') = \frac{|s \cap s'|}{\sqrt{|s| \cdot |s'|}}.$$

TABLE I
An example of $\mathcal{S}$

Set	Elements
s_1	$e_1, e_2, e_3, e_4, e_6, e_8, e_9, e_{11}, e_{12}, e_{13}, e_{15}, e_{16}, e_{19}, e_{20}$
s_2	$e_3, e_4, e_5, e_6, e_7, e_8, e_{10}, e_{14}, e_{15}, e_{16}, e_{17}, e_{19}$
s_3	$e_1, e_4, e_7, e_8, e_9, e_{10}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}, e_{20}$
s_4	$e_4, e_5, e_6, e_8, e_{10}, e_{12}, e_{14}, e_{15}, e_{16}, e_{17}, e_{19}$
s_5	$e_2, e_3, e_4, e_5, e_6, e_7, e_{11}, e_{12}, e_{15}, e_{16}, e_{17}$
s_6	$e_4, e_8, e_9, e_{11}, e_{13}, e_{14}, e_{15}, e_{16}, e_{17}, e_{18}, e_{19}$
s_7	$e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_{11}, e_{12}, e_{15}, e_{17}, e_{18}$

We use Jaccard similarity to present our algorithm in this paper. (How to handle Cosine similarity is discussed in Section IV-G.)

Given a collection $\mathcal{S}$ of sets, the set k nearest neighbor (kNN) self-join finds, for each set $s_i \in \mathcal{S}$, the k most similar sets in $\mathcal{S} \backslash \{s_i\}$ to s_i. Now we define the answer for s_i.

Definition 1 (Answer for s_i). *Given a collection $\mathcal{S}$ of sets and $k < |\mathcal{S}|$, the answer for a set s_i, $s_i.A$, satisfies $|s_i.A| = k$ and $\forall s_j \in s_i.A$, $\forall s \in \mathcal{S} \backslash \{s_i.A \cup s_i\}$, $\mathsf{Sim}(s_i, s_j) \geq \mathsf{Sim}(s_i, s)$.*

The set kNN self-join problem is formally defined as follows.

Definition 2 (Set k nearest neighbor self-join). *Given a collection $\mathcal{S}$ of sets and k, the set k nearest neighbor (kNN) self-join finds $s_i.A$ for all $s_i \in \mathcal{S}$.*

Example 1. *Assume that $k = 1$ and a set collection $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$, which is shown in TABLE I, is given. We have $\mathsf{Jac}(s_1, s_2) = \frac{7}{19}$, $\mathsf{Jac}(s_1, s_3) = \frac{9}{19}$, $\mathsf{Jac}(s_1, s_4) = \frac{7}{18}$, $\mathsf{Jac}(s_1, s_5) = \frac{8}{17}$, $\mathsf{Jac}(s_1, s_6) = \frac{8}{17}$, and $\mathsf{Jac}(s_1, s_7) = \frac{9}{18}$. Hence, $s_1.A = \{s_7\}$. Also, $s_2.A = \{s_4\}$, $s_3.A = \{s_6\}$, $s_4.A = \{s_2\}$, $s_5.A = \{s_7\}$, $s_6.A = \{s_3\}$, and $s_7.A = \{s_5\}$.*

We consider a dynamic environment where sets add new elements and remove some elements over time[1]. Hereafter, let $\mathcal{S}$ be the collection of sets consisting of all elements that have ever been generated and have not been removed. Now, we define the problem tackled in this paper.

Problem definition. *Given $\mathcal{S}$ and k, the problem of dynamic set kNN self-join is to continuously update $s.A$ for all $s \in \mathcal{S}$.*

Example 2. *Consider $\mathcal{S}$ in TABLE I and assume that s_5 has a new element e_1. Now $\mathsf{Jac}(s_1, s_5) = \frac{9}{17}$, so $s_1.A$ is updated to $\{s_5\}$ and the answers of the other sets are kept the same.*

Existing techniques. The main bottleneck of set similarity join is to compute the exact similarity between two sets. If two given sets are not similar, we should avoid pairwise similarity computation as much as possible. To this end, prior works employ *prefix-filter* and *length-filter*.

Lemma 1 (Prefix-filter [27]). *Given two sets s and s', whose elements are sorted in global order (e.g., frequency-*

[1] Our algorithm can deal with the case where a set is inserted and deleted, because this case is simply a sequence of element insertions and deletions. In addition, without loss of generality, we assume there is no set (i.e., no element) at first, and our algorithm begins when the first pair of set and element identifiers is given. In Section IV-G, we describe how our algorithm deals with the non-empty initial state.

based order), and a threshold τ, if $\mathsf{Jac}(s, s') \geq \tau$, we have $\psi(s) \cup \psi(s') \neq \varnothing$, where $\psi(s)$ is the collection of the first $(\lfloor (1-\tau)|s| \rfloor + 1)$ elements in s.

LEMMA 2 (LENGTH-FILTER [14]). *Given two sets s and s' and a threshold τ, if $\mathsf{Jac}(s, s') \geq \tau$, we have $\tau|s| \leq |s'| \leq \frac{|s|}{\tau}$.*

We also employ length-filter, since this can be computed in $O(1)$ time. Prefix-filter needs an element frequency order to strengthen its power. The order is frequently updated in our problem, which affects many sets (i.e., we need many sorting operations) and is prohibitive. Note that an experimental paper [26] has conducted extensive and comprehensive experiments to investigate the powers of existing filters. The results demonstrate that simple and light-weight filters scale better than complicated filters. Based on this observation, we propose simple, light-weight, and efficient filters[2].

III. FUNDAMENTAL PROPERTY

To efficiently process dynamic set kNN self-join, LI-DSN-Join is based on answering the questions: when a set has a new element, which sets can newly become its kNN result and whose kNN results do we have to update? We use the notations $\langle s, e\rangle_{ins}$ and $\langle s, e\rangle_{del}$ to represent that e is inserted into s and e is removed from s, respectively. (TABLE II describes the notations frequently used in this paper.) Let S_i be the collection of sets s_j where $s_i \cap s_j \neq \varnothing$ and $s_i \neq s_j$. Given $\langle s_i, e\rangle_{ins}$ or $\langle s_i, e\rangle_{del}$, $\forall s_j \in S_i$, the Jaccard similarity between s_i and s_j varies. More specifically, from the definition of Jaccard similarity, we have:

LEMMA 3. *Given $\langle s_i, e\rangle_{ins}$, the Jaccard similarity between s_i and s_j increases (decreases) iff $e \in s_j$ ($e \notin s_j$). Also, given $\langle s_i, e\rangle_{del}$, the Jaccard similarity between s_i and s_j increases (decreases) iff $e \notin s_j$ ($e \in s_j$).*

This observation is important to identify the sets whose kNN may change. When $\langle s_i, e\rangle_{ins}$ is given, s_i may newly become the kNN of sets s_j where $e \in s_j$. At the same time, s_i does not newly become the kNN of sets s_j where $e \notin s_j$. To efficiently identify the sets that have e, we employ an inverted index I, which is a collection of postings lists $I(e)$. A postings list $I(e)$ is a list of the identifiers of the sets that have e.

Next, if s_i is included in the kNN result of s_j (i.e., $s_i \in s_j.A$), $s_j.A$ and/or the threshold (i.e., the k-th largest similarity) of s_j, denoted by $s_j.\tau$, may need to be updated because of $\langle s_i, e\rangle_{ins}$ or $\langle s_i, e\rangle_{del}$ (see Example 2). Therefore, to correctly update the join result, we have to access the reverse kNN of s_i.

DEFINITION 3 (REVERSE kNN OF s_i). *Given k, s_i, and $\mathcal{S}$, $s_j \in \mathcal{S}$ is a reverse kNN of s_i if $s_i \in s_j.A$.*

For each $s_i \in \mathcal{S}$, we maintain its reverse kNN list, which consists of the identifiers of the reverse kNN of s_i. We use $s_i.RL$ to denote the reverse kNN list of s_i. This list is necessary to avoid retrieving the reverse kNN from scratch. From Lemma 4, we see the sets whose kNN may change.

TABLE II
SUMMARY OF NOTATIONS

Symbol	Description
s	A set of elements
e	An element
$\mathcal{S}$	A collection of sets
$\langle s, e\rangle_{ins}$	e is inserted into s
$\langle s, e\rangle_{del}$	e is removed from s
$s.A$	The kNN sets of s
$s.\tau$	The k-th highest similarity value (i.e., threshold) in $s.A$
$s.\tau_{temp}$	A temporal threshold during $s.A$ verification
$s.RL$	A list of the identifiers of the reverse kNN of s
S_i	The collection of sets $s_j \in \mathcal{S}$ that have $s_i \cap s_j \neq \varnothing$
I	The inverted index (collection of postings lists)
$I(e)$	A postings list of the identifiers of sets having e
$\Delta_{i,j}$	$\lvert s_i \backslash s_j \rvert$
$s_i.D$	A local-index (collection of $\langle j, \Delta_{i,j}\rangle$) of s_i

LEMMA 4. *Given $\langle s_i, e\rangle_{ins}$ and $\langle s_i, e\rangle_{del}$, the collections of sets whose kNN may change are $\{s_i\} \cup \{s_j \mid j \in s_i.RL \cup I(e)\}$ and $\{s_i\} \cup \{s_j \mid j \in s_i.RL\} \cup \{s_{j'} \mid s_{j'} \in S_i, e \notin s_{j'}\}$, respectively.*

We here introduce the space requirement of the inverted index and the reverse kNN lists, i.e., I and $\bigcup_{\mathcal{S}} s_i.RL$.

LEMMA 5. *The space complexity of the inverted index and reverse kNN lists is $O(|P| + k|\mathcal{S}|)$, where P is a collection of all pairs of set and element identifiers that have ever been generated and have not been removed.*

PROOF. The inverted index I requires $O(|P|)$ space. The reverse kNN lists require $O(\sum_{\mathcal{S}} |s_i.RL|)$ space. We have $O(\sum_{\mathcal{S}} |s_i.RL|) = O(k|\mathcal{S}|)$ from Definition 3. □

It is important to note that the cost of scanning postings lists is negligible in practice, as shown in our experiment, and the main cost of set kNN self-join is pairwise similarity computation incurred by result verification. In Section IV, we devise an algorithm to efficiently reduce this cost.

Set structure. As with [28], we use a hash table to maintain a set. The reason is twofold. First, inserting (removing) an element into (from) a set requires only $O(1)$ (expected) time. Second, the hash-based implementation provides an efficient similarity computation: given s and s', we can compute $\mathsf{Jac}(s, s')$ in $O(\min(|s|, |s'|))$ time. To further save the cost of scanning the hash table, we incorporate an early termination strategy into similarity computation. Let $\psi(s, i)$ be the collection of the first i elements in s. From Lemmas 1 and 2, we have:

LEMMA 6 (EARLY TERMINATION). *Assume $|s| \leq |s'|$, and s' cannot become the kNN of s, if $|\psi(s, i) \cap s'| + |s| - |\psi(s, i)| \leq \frac{s.\tau}{1+s.\tau}(|s| + |s'|)$.*

Lemma 6 is a generalized version of Lemma 1 and we do not need sorting any more. (A similar technique is presented in [29], but our early termination is optimized for Jaccard (and Cosine) similarity, whereas [29] focuses on inner product.) In our approach, to compute $\mathsf{Jac}(s, s')$, we need to sequentially

[2] Although existing works proposed partition-based methods, literature [5] shows that the performances of prefix-filter-based methods are better than those of partition-based methods, so here we focus on prefix-filter-based methods.

access the elements in s, and we stop the computation when s satisfies Lemma 6.

IV. LI-DSN-Join

From Lemma 4, we know sets whose kNN may change, which is an important observation to incrementally update the join result. In this section, we propose LI-DSN-Join, an algorithm that efficiently updates the kNN of them by avoiding pairwise similarity computation as much as possible.

A. Main Idea

LI-DSN-Join leverages the following ideas: (i) *the similarity of two sets s_i and s_j can be obtained in $O(1)$ time if we can obtain $|s_i \cap s_j|$ in $O(1)$ time*, (ii) *w.r.t. s_i, if $|s_i \cap s_j|$ is indexed for each (necessary) $s_j \in \mathcal{S}$, we can update the kNN of s_i, i.e., $s_i.A$, by simply scanning the index*, and (iii) *we can incrementally update $|s_i \cap s_j|$ with the inverted index I.*

Index structure. In LI-DSN-Join, each set $s_i \in \mathcal{S}$ maintains the exact difference size between s_i and s_j for each necessary set s_j (we discuss the reason why we do not use intersection size in Section IV-G). Let $\Delta_{i,j}$ be $|s_i \backslash s_j|$, and we define the local-index.

DEFINITION 4 (LOCAL-INDEX). *The local-index of s_i is a collection of pairs of j (set identifier) and $\Delta_{i,j}$ and is implemented by a hash table. We use $s_i.D$ to denote the local-index of s_i. All pairs $\langle j, \Delta_{i,j}\rangle$ such that $\Delta_{i,j} \leq \Delta_i^d$, where Δ_i^d is a threshold for the local-index, are definitely included in $s_i.D$.*

From Lemma 4 and Definition 4, we see the space complexity of LI-DSN-Join.

THEOREM 1 (SPACE COMPLEXITY). *The space complexity of LI-DSN-Join is $O(|P| + k|\mathcal{S}| + \sum_{\mathcal{S}} |s.D|)$.*

Recall that, if $\langle j, \Delta_{i,j}\rangle \in s_i.D$, we can compute $\mathsf{Jac}(s_i, s_j) = \frac{|s_i| - \Delta_{i,j}}{|s_j| + \Delta_{i,j}}$ in $O(1)$ time. Although this is promising to efficiently update the join result, the local-index provides a question: which is an appropriate Δ_i^d for exact and efficient result update? A straightforward approach is to set $\Delta_i^d = |s_i|$, but this may result in the worst space complexity, i.e., $O(|\mathcal{S}|^2)$, and is not feasible. Here, we provide a corollary, which is derived from Lemma 2, to reduce $|s_i.D|$ from $|\mathcal{S}|$ while guaranteeing correctness.

COROLLARY 1. *If $\Delta_{i,j} > (1 - s_i.\tau)|s_i|$, we have $\mathsf{Jac}(s_i, s_j) < s_i.\tau$.*

This corollary claims that if $\Delta_i^d \geq (1 - s_i.\tau)|s_i|$, we can reduce the size of the local-index $s_i.D$ and ignore all sets that are not included in $s_i.D$[3]. An intuitive approach is to specify $\Delta_i^d = \lfloor (1 - s_i.\tau)|s_i| \rfloor$, which makes $s_i.D$ compact. Recall that $|s_i|$, $s_i.\tau$, and the sizes of the other sets vary over time. The intuitive approach is too update-sensitive, and is inefficient, as demonstrated by our empirical study in Section V-B. To better understand, we give an example below.

EXAMPLE 3. *Suppose that $s_i.\tau = 0.7$ and $|s_i| = 10$. If we follow the approach, we have $\Delta_i^d = 3$. Assume that we are given $\langle s_{i'}, e\rangle_{ins}$ and the intermediate threshold (the threshold during verification) is 0.6 (i.e., we have $s_{i'} \in s_i.A$ and $e \notin s_i$). Now, sets s_j such that $\Delta_{i,j} = 4$ can be the kNN of s_i. Since $\Delta_i^d = 3$, $\langle j, \Delta_{i,j}\rangle \notin s_i.D$. Therefore, in this case, we need additional operations to guarantee correctness.*

Challenge. In the above case, we have to compute the exact difference sizes between s_i and other sets in S_i. Note that *we cannot avoid all such cases*, since $\Delta_i^d < |s_i|$, but we want to avoid this case as much as possible to reduce the update time. This renders a non-trivial challenge of determining an effective Δ_i^d. We overcome this challenge by utilizing a cost model, and this cost model is based on the operations of the kNN result update in LI-DSN-Join. Therefore, we first introduce how to update local-indices and kNN results, and then describe how to determine Δ_i^d.

B. Framework of Join Result Update

To focus on index and result update algorithms, assume that Δ_i^d satisfies $\Delta_i^d < |s_i|$. Given $\langle s_i, e\rangle_{ins}$ or $\langle s_i, e\rangle_{del}$, LI-DSN-Join updates the kNN join result with the following two steps.

1) Update the local-indices and reverse kNN lists by utilizing the inverted index I.
2) Update the kNN results of s_i and s_j for all $j \in s_i.RL$ by Δ-Scan and IF-Scan.

In Δ-Scan, for those sets whose kNN results can be updated, we verify the answers by scanning their local-indices. IF-Scan is executed only when we cannot guarantee correctness after Δ-Scan. In IF-Scan, we obtain S_i by (re-)scanning the necessary postings lists, and then determine Δ_i^d so as not to lose correctness. After that, we update the local-index, kNN result, and threshold.

C. Step 1: Index-update

In this step, we update our indices, i.e., postings lists, reverse kNN lists, and local-indices.

Observation. Let us see which $\Delta_{i,j} = |s_i \backslash s_j|$ varies when $\langle s_i, e\rangle_{ins}$ is given. If $e \in s_j$, $\Delta_{i,j}$ does not vary. Otherwise, $\Delta_{i,j}$ increases by one. On the other hand, $\Delta_{j,i}$ decreases by one if $e \in s_j$, and $\Delta_{j,i}$ does not vary otherwise. From this observation, we can identify the sets whose local-indices have to be updated. Actually, we can efficiently update them by scanning the postings list $I(e)$.

Algorithm for insertion. Algorithm 1 illustrates how to update indices when $\langle s_i, e\rangle_{ins}$ is given. Assume $e \in s_j$, i.e., $j \in I(e)$. First, if $\langle j, \cdot\rangle \notin s_i.D$ and $|s_i| - 1 \leq \Delta_i^d$, we insert $\langle j, |s_i| - 1\rangle$ into $s_i.D$ to follow Definition 4. Next, we update the local-index $s_j.D$. We have three cases: (i) $\langle i, \Delta_{j,i}\rangle \in s_j.D$, (ii) $\langle i, \Delta_{j,i}\rangle \notin s_j.D$ and $|s_j| - 1 \leq \Delta_j^d$, and (iii) otherwise.

Cases (i) and (ii) (resp. lines 6–9 and lines 10–13). In these cases, we update $s_j.D$ and/or $\Delta_{j,i}$ to follow Definition 4 and keep correctness. During this, we also update $s_i.RL$ if s_i can be kNN of s_j.

[3] The worst time and space complexities of LI-DSN-Join can be $O(|\mathcal{S}|^2)$ if all sets in $\mathcal{S}$ are similar to each other. However, this case hardly occurs in practice, and our experimental results show that LI-DSN-Join runs with fast update time and a practical memory cost on real datasets.

Algorithm 1: Index-Update (insertion)

```
   Input: $\langle s_i, e\rangle_{ins}$
1  for $\forall j \in I(e)$ do
2      /* $s_i.D$ update */
3      if $\langle j, \cdot\rangle \notin s_i.D \wedge |s_i| - 1 \leq \Delta_i^d$ then
4          $s_i.D \leftarrow s_i.D \cup \langle j, |s_i| - 1\rangle$
5      /* $s_j.D$ update */
6      if $\langle i, \Delta_{j,i}\rangle \in s_j.D$ then
7          $\langle i, \Delta_{j,i}\rangle \leftarrow \langle i, \Delta_{j,i} - 1\rangle$
8          if $s_j.\tau < \frac{|s_j|-\Delta_{j,i}}{|s_i|+\Delta_{j,i}} \wedge i \notin s_j.A$ then
9              $s_i.RN \leftarrow s_i.RN \cup \{j\}$
10     else if $|s_j| - 1 \leq \Delta_j^d$ then
11         $s_j.D \leftarrow s_j.D \cup \langle i, |s_j| - 1\rangle$
12         if $s_j.\tau < \frac{|s_j|-\Delta_{j,i}}{|s_i|+\Delta_{j,i}}$ then
13             Execute line 9
14     else
15         if $|s_i| > |s_j| - \Delta_j^d$ then
16             $\Delta_j^d \leftarrow \Delta_j^d - 1$
17             if $s_j.\tau < \frac{|s_j|-\Delta_j^d}{s_j.\tau\cdot|s_j|+\Delta_j^d}$ then
18                 Execute line 9
19 $I(e) \leftarrow I(e) \cup \{i\}$
```

Algorithm 2: Index-Update (deletion)

```
   Input: $\langle s_i, e\rangle_{del}$
1  $S' \leftarrow \bigcup_{e' \in s_i} I(e') \cup I(e)\backslash\{s_i\}$
2  for $\forall s_j \in S'$ do
3      $f \leftarrow 0$
4      if $e \in s_j$ then
5          /* $s_j.D$ update */
6          if $\langle i, \cdot\rangle \in s_j.D$ then
7              $\langle i, \Delta_{j,i}\rangle \leftarrow \langle i, \Delta_{j,i} + 1\rangle$
8      else
9          /* $s_i.D$ update */
10         if $\langle j, \Delta_{i,j}\rangle \in s_i.D$ then
11             $\langle j, \Delta_{i,j}\rangle \leftarrow \langle j, \Delta_{i,j} - 1\rangle$
12         else
13             if $f = 0$ then
14                 Update $\Delta_i^d$ as in lines 15–18 of Algorithm 1
15                 if $\Delta_i^d$ *decreases* then
16                     $f \leftarrow 1$
17         if $\langle i, \Delta_{j,i}\rangle \in s_j.D \wedge s_i \notin s_j.A$ then
18             if $s_j.\tau < \frac{|s_j|-\Delta_{i',i}}{|s_i|+\Delta_{j,i}}$ then
19                 $s_i.RN \leftarrow s_i.RN \cup \{j\}$
20 $I(e) \leftarrow I(e)\backslash\{i\}$
```

Case (iii) (lines 14–18). In this case, we do not know the exact $\Delta_{j,i}$, but can focus on the case where $|s_j| > |s_i| - \Delta_j^d$. This is because if $\Delta_j^d > \Delta_{j,i}$, we have $|s_j| > |s_i| - \Delta_j^d$ due to that fact that $\Delta_j^d > \Delta_{j,i} \geq |s_j| - |s_i|$. Note that now $\Delta_{j,i}$ may be Δ_j^d. Therefore, we decrement Δ_j^d by one to avoid computing the exact $\Delta_{j,i}$ while satisfying the condition in Definition 4. We then compute an upper-bound of Jac(s_j, s_i), which is shown in line 17 and derived from Lemma 2 and $\Delta_{j,i} \geq \Delta_j^d$. If this upper-bound exceeds $s_j.\tau$, s_i can be the kNN of s_j, so $\{j\}$ is inserted into $s_i.RL$.

For all $j \in I(e)$, the above operations are executed. After that, we insert $\{i\}$ into $I(e)$ (line 19). We see that all lines in Algorithm 1 need a constant time, thus the time complexity of Algorithm 1 is $O(|I(e)|)$.

Algorithm for deletion. We take a similar procedure when $\langle s_i, e\rangle_{del}$ is given. Algorithm 2 illustrates how to update the indices when $\langle s_i, e\rangle_{del}$ is given. The difference between Algorithms 1 and 2 is twofold. The first is that $\Delta_{i,j}$ decreases if $e \notin s_j$ and $\Delta_{i,j}$ does not vary otherwise. Also, $\Delta_{j,i}$ increases if $e \in s_j$ and $\Delta_{j,i}$ does not vary otherwise. This difference renders the second difference: we scan $S' = \{s_j \,|\, s_j \in \mathcal{S}, s_i \cap s_j \neq \varnothing \vee e \in s_j\}$. Hence, the index update algorithm for dealing with $\langle s_i, e\rangle_{del}$ requires $O(|S'|)$ time.

Remark. Note that deletions in the applications introduced in Section I are very rare compared with insertions [7], [30]. Therefore, in practice, we can update the indices by a single scanning a postings list, meaning that we can deal with this step quickly. Our empirical study in Section V-C shows that the main cost of join result update is kNN result updates.

D. Step 2-1: Δ-Scan

Overview. In this step, for each set s whose kNN may change, we verify the answer by using its local-index. In a nutshell, to update the kNN of s, we first obtain a temporal threshold from the previous kNN result, and then verify its answer by leveraging $s.D$ and/or Lemma 3.

Algorithm description. Algorithm 3 describes how to update $s.A$ when $\langle s_i, e\rangle$, which is $\langle s_i, e\rangle_{ins}$ or $\langle s_i, e\rangle_{del}$, is given.

Case $s = s_i$. We take the following steps.

1) At lines 2–7, we obtain a temporal threshold, $s_i.\tau_{temp}$, from the previous kNN result $s_i.A_{prev}$. During this, the local-index $s_i.D$ is also updated to keep correctness.
2) Next, at lines 8–11, we scan $s_i.D$ to verify $s_i.A$ while updating the corresponding reverse kNN lists.
3) Last, lines 12–15 confirm whether we need IF-Scan (its detail is explained in Section IV-E). If yes, we execute IF-Scan. Otherwise, we set $s_i.\tau = s_i.\tau_{temp}$ and terminate updating $s_i.A$.

Case $s = s_j$ ($j \in s_i.RL$). In this case, we update the kNN results of sets s_j where s_j is (or can be) a reverse kNN of s_i. We have two patterns: $s_i \in s_j.A$ and $s_i \notin s_j.A$ (j has been newly inserted into $s_i.RL$ by the index update algorithm).

- If $s_i \in s_j.A$ (lines 18–22), we compute Jac(s_j, s_i), update $s_i.\tau$, and then confirm whether $s_j.\tau$ decreases or not. From Lemma 3, we do not need to scan $s_j.D$ if $s_j.\tau$ does not decrease and the update is over. On the other hand, if $s_j.\tau$ decreases, we take operations (2) and (3) of case $s = s_i$.
- If $s_i \notin s_j.A$ and $\langle i, \Delta_{j,i}\rangle \in s_j.D$ (lines 24–27), we see that s_i certainly becomes the kNN of s_j. Therefore, we simply update $s_j.A$, $s_j.\tau$, and the corresponding reverse kNN lists. If $s_i \notin s_j.A$ and $\langle i, \Delta_{j,i}\rangle \notin s_j.D$, we execute IF-Scan. (For example, this situation corresponds to line 17 of Algorithm 1.)

E. Step 2-2: IF-Scan

If some sets, which are not indexed in the local-indices, can be the kNN results, we execute IF-Scan. Assume that

Algorithm 3: Δ-Scan

```
   Input: $\langle s_i, e\rangle$, $s$
1  case $s = s_i$ do
2      $s_i.\tau_{temp} \leftarrow 1$, $s_i.A_{prev} \leftarrow s_i.A$
3      for $\forall\langle j, \Delta_{i,j}\rangle \in s_i.D$ s.t. $j \in s_i.A_{prev}$ do
4          if $\langle s_i, e\rangle = \langle s_i, e\rangle_{ins} \wedge e \notin s_j$ then
5              $\langle j, \Delta_{i,j}\rangle \leftarrow \langle j, \Delta_{i,j} + 1\rangle$
6          if $s_i.\tau_{temp} > \frac{|s_i|-\Delta_{i,j}}{|s_j|+\Delta_{i,j}}$ then
7              $s_i.\tau_{temp} \leftarrow \frac{|s_i|-\Delta_{i,j}}{|s_j|+\Delta_{i,j}}$
8      for $\forall\langle j, \Delta_{i,j}\rangle \in s_i.D$ s.t. $j \notin s_i.A_{prev}$ do
9          Execute lines 4–5
10         if $\frac{|s_i|-\Delta_{i,j}}{|s_j|+\Delta_{i,j}} > s_i.\tau_{temp}$ then
11             Update $s_i.A$, $s_i.\tau_{temp}$, $s_j.RL$, and $s_{j'}.RL$   ▷ $s_{j'}$ is
                the previous $k$NN of $s_i$
12     if $(1 - s_i.\tau_{temp})|s_i| > \Delta_i^d$ then
13         IF-Scan
14     else
15         $s_i.\tau \leftarrow s_i.\tau_{temp}$
16 case $s \neq s_i$ ($s = s_j$) do
17     if $s_i \in s_j.A$ then
18         if $\frac{|s_j|-\Delta_{j,i}}{|s_i|+\Delta_{j,i}} \geq s_j.\tau$ then
19             Update $s_j.\tau$
20         else
21             $s_j.\tau_{temp} \leftarrow \frac{|s_j|-\Delta_{j,i}}{|s_i|+\Delta_{j,i}}$
22             Execute lines 8–15 (w/o line 9)
23     else
24         if $\langle i, \Delta_{j,i}\rangle \in s_j.D$ then
25             Update $s_j.A$, $s_j.\tau$, $s_i.RN$, and $s_{j'}.RN$
26         else
27             IF-Scan
```

IF-Scan is executed for s_i. IF-Scan is a simple operation:

1) We determine Δ_i^d by our proposed cost model, and then
2) We update $s_i.D$ while updating $s_i.A$ and $s_i.\tau$, by obtaining all $\langle j, \Delta_{i,j}\rangle$ such that $\Delta_{i,j} \leq \Delta_i^d$.

We see that the main challenge here is how to determine Δ_i^d. We below overcome this challenge.

Cost model for determining Δ_i^d. Recall that as long as we have $(1 - s_i.\tau)|s_i| \leq \Delta_i^d$, Δ-Scan requires only $O(|s_i.D|)$ time. On the other hand, in IF-Scan, we have to retrieve a collection S_i of sets s_j such that $s_i \cap s_j \neq \varnothing$ and then compute $\Delta_{i,j}$ for every $s_j \in S_i$ such that $\langle j, \cdot\rangle \notin s_i.D$. This requires

$$O(\sum_{e' \in s_i} |I(e')| + (|S_i| - |s_i.D|)|s_i|) \tag{1}$$

time. (We employ Lemmas 2 and 6 to save practical computational cost.) These time complexities suggest that we should avoid IF-Scan as much as possible. Therefore, we set

$$\Delta_i^d = \lfloor(1 - s_i.\tau)|s_i|\rfloor + \alpha, \tag{2}$$

where α is a non-negative integer. If α is large, we have less situations which need IF-Scan, because $s_i.D$ can have many $\Delta_{i,j}$. However, Δ-Scan becomes heavy since $|s_i.D|$ becomes large. We therefore propose a cost model and obtain α that minimizes the expected cost of updating the kNN result of s_i. To enable the cost model design, we put the four following assumptions.

1. For any two $\Delta_{i,j}$ and $\Delta_{i,j'}$, we have $\Delta_{i,j} \neq \Delta_{i,j'}$: Let $s_i.D_\alpha$ be $s_i.D$ satisfying Definition 4 based on Equation (2). This provides that $|s_i.D_{\alpha+1}|$ is at most $|s_i.D_\alpha| + 1$.
2. Each set is updated uniformly at random[4]: Given $\langle s, e\rangle$, the probability that $s = s_i$ is $\frac{1}{|\mathcal{S}|}$.
3. Given $\langle s, e\rangle$, $s_i.\tau$ does not vary: The thresholds of sufficiently large sets vary only a little (see Example 2).
4. Whenever $\langle s_j, e\rangle$, where $\langle j, \cdot\rangle \notin s_i.D$, is given, Δ_i^d is decremented by one: As can be seen from Algorithm 1, this assumes the worst scenario.

Now define

$$f(\alpha) = cost_\Delta(\alpha) + cost_{IF}(\alpha),$$

where $cost_\Delta(\alpha)$ and $cost_{IF}(\alpha)$ respectively be the expected costs of Δ-Scan and IF-Scan for s_i when α is given. The case where Δ-Scan is executed for s_i can be that $s = s_i$ and $s = s_j$ where $s_j \in s_i.A$. That is, the probability of executing Δ-Scan for s_i is $\frac{k+1}{|\mathcal{S}|}$. Therefore, we have

$$cost_\Delta(\alpha) = (\gamma + \alpha)\frac{k+1}{|\mathcal{S}|},$$

where $\gamma = |\{j \mid \langle j, \Delta_{i,j}\rangle \in s_i.D, \Delta_{i,j} \leq (1 - s_i.\tau_{temp})|s_i|)\}|$. Note that γ is an estimated minimum number of $\Delta_{i,j}$ that satisfies the condition in Definition 4 when we have $(1 - s_i.\tau_{temp})|s_i|) > \Delta_i^d$. From assumption 1, we can use $\gamma + \alpha$ as an estimated size of $s_i.D$ after Δ_i^d is updated in IF-Scan. (Recall that the above cost computation is executed in IF-Scan.) To obtain the exact number of $\Delta_{i,j}$ that satisfies the condition in Definition 4, we have to compute $\Delta_{i,j}$ for all $s_j \in S_i$ such that $\langle j, \Delta_{i,j}\rangle \notin s_i.D$. This is clearly expensive, so we use the estimation, which requires just the scanning of $s_i.D$. Next, the probability that Δ_i^d is decremented corresponds to the probability that $s = s_j$ such that $\langle j, \Delta_{i,j}\rangle \notin s_i.D$, i.e., $\frac{|S_i|-(\gamma+\alpha)}{|\mathcal{S}|}$. If this occurs $(\alpha+1)$ times, we see, from Equation (2), that we would have $(1 - s_i.\tau_{temp})|s_i| > \Delta_i^d$. Therefore, from (1),

$$\begin{aligned}&cost_{IF}(\alpha)\\&= (\sum_{e' \in s_i} |I(e')| + (|S_i| - |s_i.D|)|s_i|)(\frac{|S_i| - \gamma - \alpha}{|\mathcal{S}|})^{\alpha+1}.\end{aligned}$$

We would like to obtain α^* that satisfies

$$\alpha^* = \underset{0 \leq \alpha \leq |S_i| - \gamma}{\arg\min} f(\alpha). \tag{3}$$

Note that $f(\alpha)$ is downward convex, i.e., as α increases, $f(\alpha)$ decreases and then increases.

[4] This assumption may not hold for real datasets, but actually, our cost model is orthogonal to any probability distribution. That is, if we know the distribution in advance, applications just change $cost_\Delta(\alpha)$ and $cost_{IF}(\alpha)$ accordingly. However, assuming that the distribution is pre-known is also not practical. We therefore use the uniform distribution by default, and our experimental results show that this setting does not lose the efficiency of our algorithm.

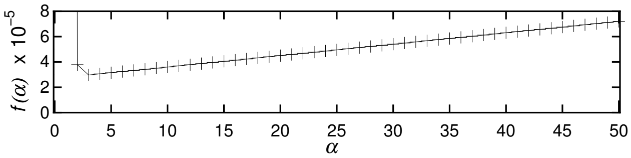


Fig. 1. Example of how $f(\alpha)$ varies when $k = 8$, $|\mathcal{S}| = 1 \times 10^6$, $|s_i| = 100$, $|S_i| = 1 \times 10^3$, $\gamma = 30$, and $\sum_{e' \in s_i} |I(e')| = 3 \times 10^3$

EXAMPLE 4. *Fig. 1 shows how $f(\alpha)$ varies in the case where $k = 8$, $|\mathcal{S}| = 1000000$, $|s_i| = 100$, $|S_i| = 1000$, $\gamma = 30$, and $\sum_{e' \in s_i} |I(e')| = 3000$. We see that $f(0)$, which shows the case that we set $\Delta_i^d = (1 - s_i.\tau)|s_i|$, incurs a very high cost. In this example, $\alpha^* = 3$.*

Although the time complexity of computing Equation (3) is $O(|S_i|)$, we can quickly obtain α^*. We see that α^* is a small positive integer in practice, and we can stop computing Equation (3) when we have $f(\alpha - 1) < f(\alpha)$, due to the downward convex property.

F. Theoretical Analysis

In this section, we first prove the correctness of LI-DSN-Join. After that, we show the efficiency of LI-DSN-Join, by comparing with its possible variants.

Correctness. We discuss the correctness of LI-DSN-Join through Theorem 2.

THEOREM 2. *LI-DSN-Join monitors the exact join result.*

PROOF. If $s.D$ is correct, it is trivial that Δ-Scan and IF-Scan compute the correct answer from Corollary 1. A non-trivial point is that we always have $\Delta_{i,j} = |s_i| - 1$ when $|s_i| - 1 \leq \Delta_i^d$ and $\langle j, \cdot \rangle \notin s_i.D$ (e.g., see line 3 of Algorithm 1). We prove this.

Consider the insertion-only case. When we first evaluate $\Delta_{i,j}$, we have $|s_i| = 1$ or $|s_j| = 1$, thereby it is trivial that $\Delta_{i,j} = |s_i| - 1$. Assume $\langle j, \Delta_{i,j} \rangle \notin s_i.D$, i.e., at a certain time when $\langle s_j, e \rangle_{ins}$ is given, we had $\Delta_{i,j} > \Delta_i^d$. Given $\langle s_i, e \rangle_{ins}$, $\Delta_{i,j}$ does not vary. If Δ_i^d has not varied or decreased, we have $\Delta_{i,j} > \Delta_i^d$. Note that the case where Δ_i^d has increased means that we have executed IF-Scan for s_i, i.e., $\Delta_{i,j}$ was exactly evaluated. We thus have $\Delta_{i,j} > \Delta_i^d$. Therefore, LI-DSN-Join keeps the correct $s_i.D$ in the insertion-only case.

Next, we consider that deletions also occur. From the above discussion, if we have $\langle j, \Delta_{i,j} \rangle \notin s_i.D$ before the first deletion occurs, we have $\Delta_{i,j} > \Delta_i^d$. Assume that $\langle s_i, e \rangle_{del}$, where $e \notin s_j$, is the first deletion for s_i. In this case, Δ_i^d may be decremented by one (see Algorithm 2).

- If Δ_i^d is not decremented, the discussion of case (iii) in index-update guarantees that $\Delta_{i,j} > \Delta_i^d$ still holds.
- If Δ_i^d is decremented by one and this incurs IF-Scan, we can keep $s_i.D$ correct.
- Even if the decrement does not incur IF-Scan, we have $\Delta_{i,j} > \Delta_i^d$. Before $\langle s_i, e \rangle_{del}$ is given, we had $\Delta_{i,j} - 1 \geq \Delta_i^d$, and the decrement derives $\Delta_{i,j} - 1 > \Delta_i^d - 1$, so the above claim is true.

We keep the correctness of $s_i.D$ even if deletions occur. □

Efficiency against variants. We next theoretically show that the simple design of LI-DSN-Join allows better performance than its possible variants.

• *Comparing with a variant minimizing $s.D$.* Given $\langle s_i, e \rangle_{ins}$ or $\langle s_i, e \rangle_{del}$, the main overhead of the join result update derives from updating the kNN result of s_i, because the kNN results of s_j, where $j \in s_i.RL$, rarely change in practice[5]. As long as we have $(1 - s_i.\tau)|s_i| \leq \Delta_i^d$, the update cost is $O(|s_i.D|)$, since our algorithm needs to scan all elements in $s_i.D$ to update its kNN sets. In the following discussion, we assume that we have $(1 - s_i.\tau)|s_i| \leq \Delta_i^d$. If we can minimize $|s_i.D|$, the cost of the kNN result update can be optimal (since each similarity computation is already optimal, i.e., $O(1)$). The k-skyband approach achieves this [31].

DEFINITION 5 (DOMINANCE). *Given s_i, s_j, and $s_{j'}$, we say that s_j dominates $s_{j'}$ w.r.t. s_i, if $|\Delta_{i,j}| \leq |\Delta_{i,j'}| \wedge |s_j| < |s_{j'}|$ or $|\Delta_{i,j}| < |\Delta_{i,j'}| \wedge |s_j| \leq |s_{j'}|$.*

DEFINITION 6 (k-SKYBAND). *Given s_i and $\mathcal{S}$, the k-skyband of s_i is a collection of $\langle j, \Delta_{i,j} \rangle$ where $s_j \in \mathcal{S}$ is dominated by at most $(k - 1)$ other sets $\in \mathcal{S} \backslash \{s_i\}$ w.r.t. s_i.*

It is important to note that s_j, such that $\langle j, \Delta_{i,j} \rangle$ is not in the k-skyband of s_i, cannot be the kNN of s_i, because there are at least k sets such that $\mathsf{Jac}(s_i, s_j) > \mathsf{Jac}(s_i, s_{j'})$ in the k-skyband. That is, the k-skyband of s_i is a minimized version of $s_i.D$ to monitor the exact result. One may consider that we should use this approach to obtain better performance. However, this approach cannot beat LI-DSN-Join.

THEOREM 3. *LI-DSN-Join outperforms the approach minimizing $s.D$.*

PROOF. The efficiency has to be measured by summation of index update and result update costs. Assume that we have executed Algorithm 1 (or 2), and focus on s_i. (Algorithms 1 and 2 can be common with LI-DSN-Join and the approach minimizing $s.D$.) LI-DSN-Join updates $s_i.A$ and $s_i.D$ at the same time (see steps (1) and (2) of Δ-Scan), which requires $O(|s_i.D|)$. Let $s_i.D_{sky}$ be the k-skyband of $s_i.D$, and we show that the update cost of $s_i.D_{sky}$ exceeds $O(|s_i.D|)$.

First, even if we use the k-skyband, $s_i.D$ has to be maintained. If $|s_j|$, where $\langle j, \Delta_{i,j} \rangle \in s_i.D_{sky}$, increases, non k-skyband sets of s_i can newly be in its k-skyband. Without $s_i.D$, we have to retrieve such sets from scratch, which costs more than $O(|s_i.D|)$ trivially. So, assume that the k-skyband approach also maintains $s_i.D$. Given a dataset with cardinality n, the time complexity of updating the k-skyband of the dataset is at least $O(\beta \log n + n_{sky})$, where β and n_{sky} are the number of data, whose elements are updated, and the size of k-skyband [32]. In our case, $\beta = O(|s_i.D|)$ and $n = |s_i.D|$, thereby the time complexity of updating k-skyband is $O(|s_i.D| \log |s_i.D| + |s_i.D_{sky}|)$, which is more than $O(|s_i.D|)$. □

• *Comparing with a variant incorporating an early termination strategy.* Now, the only remaining approach to reduc-

[5] In our experiments, we observed that the ratio of changing kNN results due to a single element insertion/ deletion is less than 0.01.

TABLE III
DATASET STATISTICS (l_{max} AND l_{avg} RESPECTIVELY MEAN THE MAXIMUM AND AVERAGE SIZE OF A SET IN $\mathcal{S}$)

Dataset	Type	Set	Element	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{E}\rvert$	l_{max}	l_{avg}
Amazon [33]	Rating	User	Product	2,385,033	5,523,611	4,143	4.4
Book [33]	Rating	User	Book	1,677,240	4,685,112	9,479	6.0
Delicious [34]	Folksonomy	User	Tag	641,224	5,220,144	2,331	15.6
Epinions [34]	Rating	User	Product	73,122	414,714	82,022	68.4
Flickr [34]	Relationship	User	User (Friend)	925,756	1,073,148	11,538	10.8
LiveJournal [35]	Relationship	User	Group	2,608,963	2,607,698	1,188	3.8
Netflix [36]	Rating	Movie	User	15,407	252,676	11,270	322.4
Orkut [35]	Relationship	User	Group	2,267,263	2,290,427	1,474	4.4
Patent [34]	Citation	Patent	Patent	1,989,695	2,774,910	472	5.0
Pokec [34]	Affiliation	User	Group	1,219,118	1,253,328	2,930	8.2

ing the cost of updating the kNN result of s_i is an early termination strategy. There is an algorithm that enables this. Given s_i, if $\Delta_{i,j} = \Delta_{i,j'}$ and $|s_j| < |s_{j'}|$, $\mathsf{Jac}(s_i, s_j) > \mathsf{Jac}(s_i, s_{j'})$. Hence, if $\mathsf{Jac}(s_i, s_j) \leq s_i.\tau$, we can ignore $s_{j'}$. That is, if we maintain the set sizes in non-decreasing order for $\{\Delta_{i,j}, ..., \Delta_{i,j'}\}$ where $\Delta_{i,j} = \cdots = \Delta_{i,j'}$, we enable early termination. However, the cost of updating this structure is $O(|s_i.D| \log |s_i.D|)$, meaning that LI-DSN-Join outperforms this approach.

Remark. The above analysis verifies that the index update and result update costs of LI-DSN-Join are well balanced. In other words, the indices employed in LI-DSN-Join is simple yet effective and also renders efficient join update. In Section V, we empirically show the efficiency.

G. Discussion

Cosine similarity case. First, we introduce prefix- and length-filters for Cosine similarity [26].

LEMMA 7 (PREFIX-FILTER). *Given two sets s and s', whose elements are sorted in global order, and a threshold τ, if $\mathsf{Cos}(s, s') \geq \tau$, we have $\psi(s) \cup \psi(s') \neq \varnothing$, where $\psi(s)$ is the collection of the first $(\lfloor(1-\tau^2)|s|\rfloor + 1)$ elements in s.*

LEMMA 8 (LENGTH-FILTER). *Given two sets s and s' and a threshold τ, if $\mathsf{Cos}(s, s') \geq \tau$, we have $\tau^2|s| \leq |s'| \leq \frac{|s|}{\tau^2}$.*

From the above lemmas, we have the following lemma and corollary.

LEMMA 9 (EARLY TERMINATION). *Assume $|s| \leq |s'|$, and s' cannot become the kNN of s for Cosine similarity, if $|\psi(s,i) \cap s'| + |s| - |\psi(s,i)| \leq s.\tau\sqrt{|s| \cdot |s'|}$.*

COROLLARY 2. *If $\Delta_{i,j} > (1 - s_i.\tau^2)|s_i|$, we have $\mathsf{Cos}(s_i, s_j) < s_i.\tau$.*

Our algorithm handles Cosine similarity from the above results, and the fact that $\mathsf{Cos}(s_i, s_j) = \frac{|s_i| - \Delta_{i,j}}{|s_i| \cdot |s_j|}$. Furthermore, the upper-bounding in line 17 of Algorithm 1 is replaced by $\frac{|s_j| - \Delta_j^d}{\sqrt{s_j.\tau|s_j|}}$. We finally note that $s_i.\tau$ in Equation (2) is replaced by $s_i.\tau^2$.

Why we do not use intersection size. If we use intersection size, we have to use 1 as a threshold for the local-index to guarantee correctness. This setting never provides a situation that we can tune the threshold, because each set s_i certainly indexes s_j for all $s_j \in S_i$. This approach incurs $O(|\mathcal{S}|^2)$ space, which is prohibitive.

Non-empty initial state. We here assume that all sets have their kNN results. To construct an inverted index I, we simply scan all sets. Also, to construct reverse kNN lists, simply scanning the kNN sets for each set is enough. LI-DSN-Join needs to construct local-index for each set. We can achieve this by executing $\mathsf{IF\text{-}Scan}$ for each set.

Batch update. We assume that set update appears one by one, but some applications may provide a collection of set updates in a batch. In this case, an approach, which runs Algorithm 1 or 2, for each set update in the collection, and then runs Algorithm 3, can be considered. One may consider that this approach decreases the number of executing Algorithm 3, but actually have little (or no) gain against the one that executes LI-DSN-Join for each set update in the batch. This is because we have to execute lines 2–7, line 9, and lines 16–27 of Algorithm 3 for each set which has new/expired elements. We see that this approach is essentially the same as LI-DSN-Join for a single update. (We empirically show this observation in Apeendix A.) To summarize, optimizing LI-DSN-Join for efficient batch update is not trivial, thus we leave it for future work.

V. EMPIRICAL STUDY

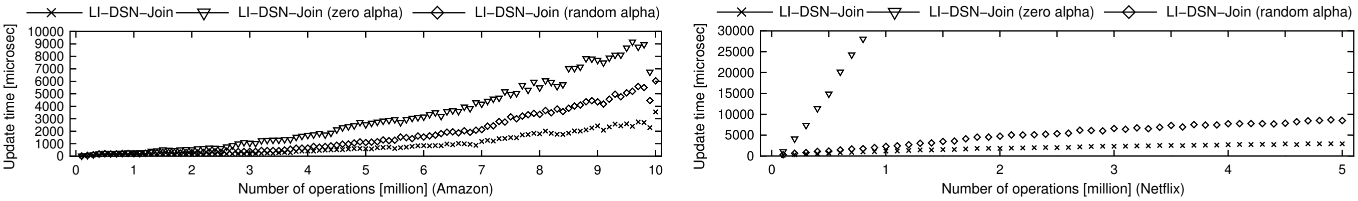
This section presents our experimental study. All experiments were conducted on a machine with Intel Xeon E5-2687W v4 processors (3.0GHz) and 512GB RAM. We report the results of experiments which used Jaccard similarity as a similarity function.

A. Setting

Datasets. We used 10 real datasets, Amazon, Book, Delicious, Epinions, Flickr, LiveJournal, Netflix, Orkut, Patent, and Pokec. We provided 5 million set updates (including both insertions and deletions) for Netflix and Epinions and 10 million updates for the others[6]. Note that the tuples of set and element identifiers are sorted by their generation time order. Unfortunately, some datasets miss the generation time, so we used random order for them [7] (note that this random generation order keeps the set update probability distributions of real datasets).

We did not generate deletions in the first 1 million updates, and we varied the percentage of deletions in our experiments.

[6] ALL did not terminate each experiment within a month when we provided 10 million updates in Netflix and Epinions cases.



(a) Amazon

(b) Netflix

Fig. 2. Scalability of LI-DSN-Join, LI-DSN-Join (zero alpha), and LI-DSN-Join (random alpha) ($k = 8$ and $\epsilon = 0.001$)

Let ϵ be the percentage of deletions, and the updates have approximately $1 \times 10^6 + (1 - \epsilon) \times 9 \times 10^6$ insertions and $\epsilon \times 9 \times 10^6$ deletions. As mentioned in Section IV-B, deletions are rare, so ϵ is a very small value. When a given update is deletion, we removed the oldest element from a given set. TABLE III shows their statistics with no deletions.

Algorithms. Since this is the first work of the dynamic kNN set similarity self-join problem, there is no existing algorithm. Therefore, we compared LI-DSN-Join with a state-of-the-art exact set similarity join algorithm ALL [14] and a state-of-the-art top-k set similarity search algorithm Tree [6]. We applied Lemmas 3 and 4 to ALL and Tree. For example, in Tree, Lemma 4 suggests which set should be considered as a query, and we specify all sets whose kNN may change. The above algorithms were implemented in C++.

We do not consider other algorithms for exact set similarity join, such as PPJ [2], [14] and SKJ [5], that rely on pre-sorting order and pre-processing approaches. They are obviously inappropriate to our dynamic environment. We do not consider the algorithm SizeAware [28]. SizeAware solves the problem of overlap set similarity join (i.e., given an intersection threshold z, this problem finds all pairs $\langle s, s' \rangle$ such that $|s \cap s'| \geq z$), but [28] shows that SizeAware can deal with Jaccard similarity. Given z, SizeAware (basically) enumerates all subsets whose sizes are z for each set and finds pairs of sets that have the same subset. Recall that our problem does not know the threshold in advance. If we apply SizeAware to our problem, it has to enumerate subsets of all sizes in $[1, |s|]$ (and actually set pairs with the same subsets are not final result but the candidates of it [28]). We see that a subset with size 1 is a single element. ALL finds pairs of sets that have the same element but does not need to enumerate subsets with the size ≥ 2. SizeAware is therefore always less efficient than ALL. Furthermore, an experimental paper [26] demonstrates that ALL shows good (often best) performance over many real datasets, which also motivated us to use ALL. Although SpotSigs [37], which is an inverted index based algorithm as well as ALL, was also tested, we do not show its result, because its performance is worse than that of ALL.

B. Cost model evaluation

We evaluate the effectiveness of our α selection by comparing with LI-DSN-Join that employs zero and random α, denoted by LI-DSN-Join (zero alpha) and LI-DSN-Join (random alpha), respectively. Due to the space limitation, we show the results on Amazon and Netflix ($k = 8$ and $\epsilon = 0.001$), which are illustrated in Fig. 2. Each plot shows the average update time every 100,000 updates.

First, we see that LI-DSN-Join (zero alpha) does not scale well. In particular, on Netflix, LI-DSN-Join (zero alpha) shows poor performance, because it executes IF-Scan many times. This result validates our cost model and demonstrates the effectiveness of selecting non-zero α. Next, randomly choosing α cannot outperform LI-DSN-Join, which verifies the effectiveness of our α selection. That is, our cost model in LI-DSN-Join successfully controls the local-indices (α more specifically) to incur less IF-Scan executions. On Amazon and Netflix, compared with LI-DSN-Join (random alpha), LI-DSN-Join achieves respectively 48.1% and 58.7% reduction w.r.t. join update time on average.

C. Index update time

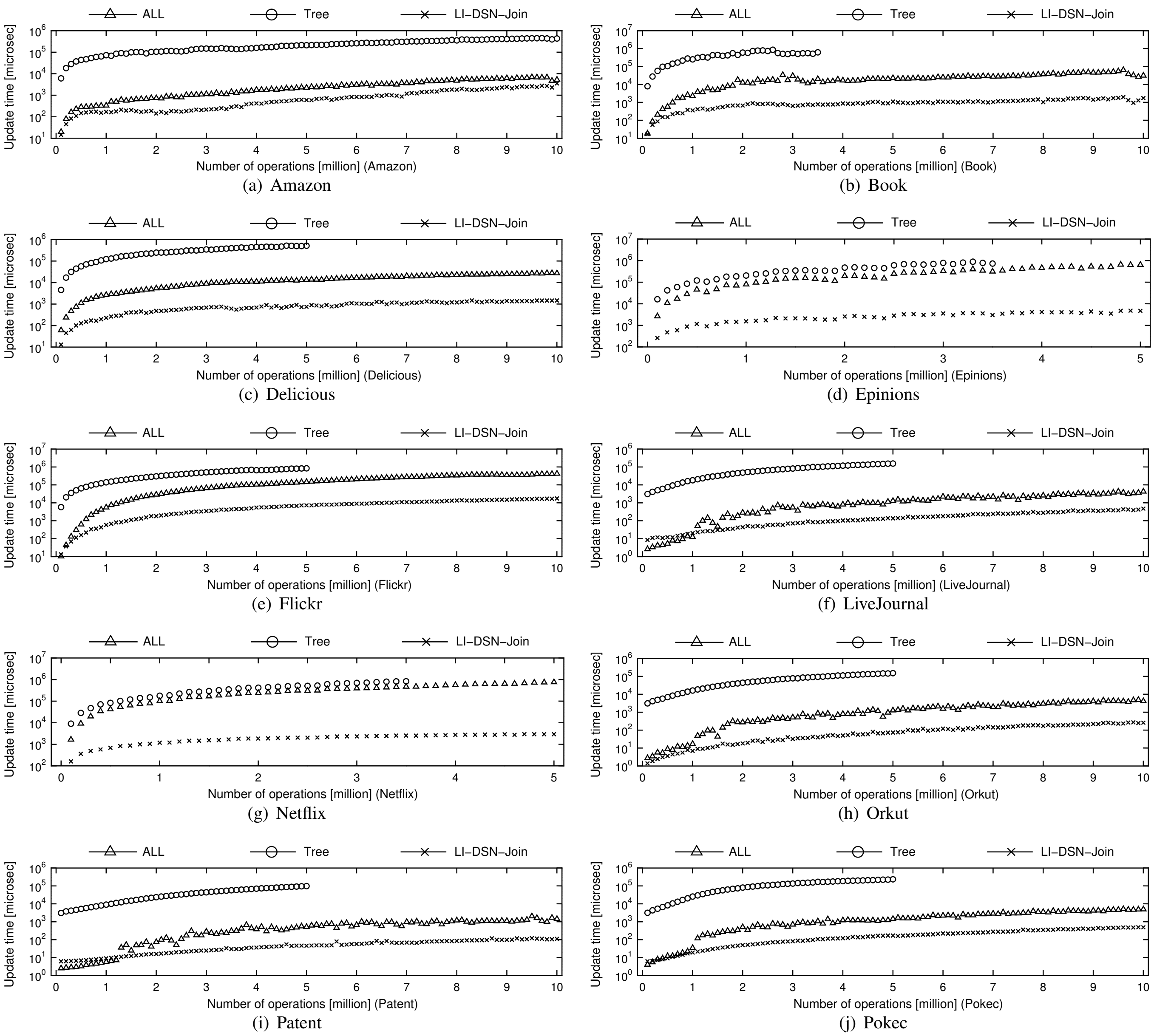
We next demonstrate that the cost of scanning postings lists is negligible in practice, which is claimed in Section III. Recall that Algorithms 1 and 2 correspond to scanning postings lists. We therefore show that the ratio of execution time of Algorithm 1 or Algorithm 2 to the total join update time of LI-DSN-Join when $\langle s, e \rangle_{ins}$ or $\langle s, e \rangle_{del}$ is given. This experiment used Amazon, LiveJournal, Netflix, and Orkut.

The average ratios of the scanning time of postings lists to the join result update time of LI-DSN-Join ($k = 8$ and $\epsilon = 0.001$) in Amazon, LiveJournal, Netflix, and Orkut are 1.88×10^{-2}, 1.09×10^{-1}, 9.57×10^{-3}, and 9.21×10^{-2}, respectively. We see that the ratios are small, which confirms that the main cost of join result update is kNN result verification.

D. Comparison with Existing Techniques

We compared LI-DSN-Join with ALL and Tree. Fig. 3 illustrates how the efficiency of each algorithm changes as sets are updated in the setting where $k = 8$ and $\epsilon = 0.001$. We see that the update time of all the algorithms basically increases as sets are updated. Furthermore, we have two main observations. (i) LI-DSN-Join scales well and outperforms the other algorithms. (ii) Scan-based approach (i.e., LI-DSN-Join and ALL) is better than the tree traversal-based approach. We terminated the experiments on Tree before it deals with 10 (5) million updates, because Tree is clearly outperformed by the algorithms. Note that we obtained a similar result when we used Cosine similarity (see Appendix B).

The observation (i) demonstrates the efficiency and scalability of our approach. Compared with ALL and Tree, LI-DSN-Join can update the join result incrementally, thanks to the efficient index update (Algorithms 1 and 2), Δ-Scan, and



(a) Amazon

(b) Book

(c) Delicious

(d) Epinions

(e) Flickr

(f) LiveJournal

(g) Netflix

(h) Orkut

(i) Patent

(j) Pokec

Fig. 3. Scalability of ALL, Tree, and LI-DSN-Join ($k = 8$ and $\epsilon = 0.001$)

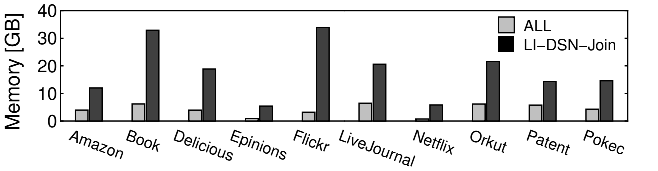
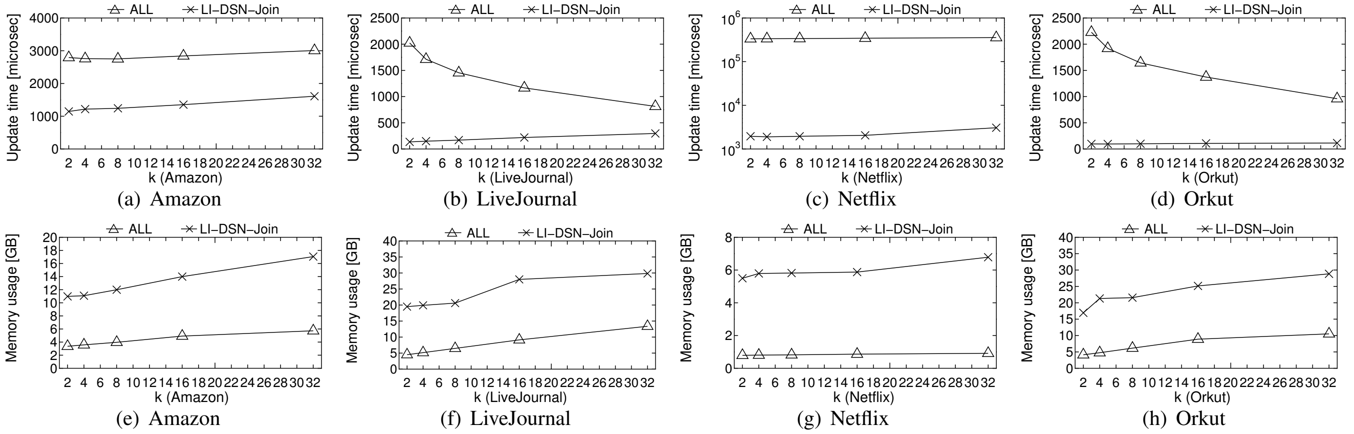


Fig. 4. Memory usage in GB ($k = 8$ and $\epsilon = 0.001$)

the cost model which has been verified in Section V-B. For example, LI-DSN-Join can process 10 (5) million updates 2, 24, 16, 98, 23, 9, 202, 16, 11, 10 times faster than ALL on Amazon, Book, Delicious, Epinions, Flickr, LiveJournal, Netflix, Orkut, Patent, and Pokec, respectively. From this result, we see that LI-DSN-Join is particularly efficient when a given set collection has large l_{avg} (e.g., Epinions and Netflix). This is derived from the local-index, i.e., the similarity between two sets is obtained in $O(1)$ time.

The observation (ii) is also an interesting result, and actually, a related work, inner product join, also has a similar observation [29], [38]. For our problem, scan-based approach, which uses for example an inverted index, shows superior performance to tree-based one. As shown in [26], light-weight filters are suitable for set similarity join, and Tree does not pay off the upper-bounding costs at intermediate nodes. Besides, for a given set, selecting candidates of its kNN sets by an inverted index (i.e., ALL) is easy and efficient, compared with Tree. Note that LI-DSN-Join is also categorized into the scan-based approach.

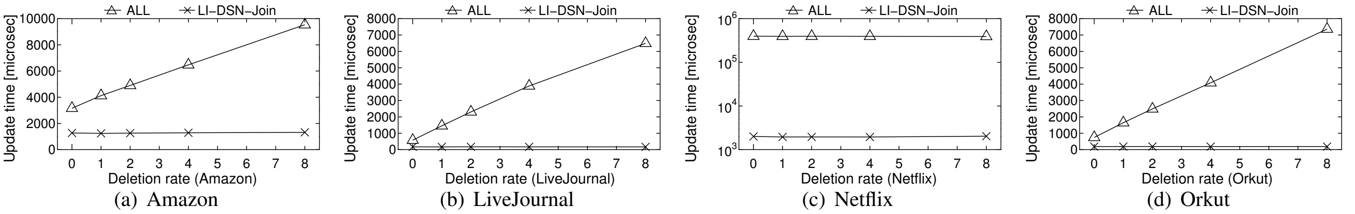
Fig. 4 shows the memory usage of ALL and LI-DSN-Join. We omit the memory usage of Tree, since, as mentioned above, we broke off the experiments on Tree. LI-DSN-Join uses more memory than ALL because of local-indices. The difference between ALL and LI-DSN-Join w.r.t. memory usage tends to



(a) Amazon (b) LiveJournal (c) Netflix (d) Orkut

(e) Amazon (f) LiveJournal (g) Netflix (h) Orkut

Fig. 5. Average update time and memory usage vs. k ($\epsilon = 0.001$)



(a) Amazon (b) LiveJournal (c) Netflix (d) Orkut

Fig. 6. Average update time vs. deletion rate ($k = 8$)

be larger when a given dataset has larger $|\mathcal{S}|$ and/or l_{max}. Each set in $\mathcal{S}$ employs a local-index. Also, sets that have many elements share same elements with many sets, so their local-indices tend to be larger. Therefore, this result is obtained. It is important to note that the memory usage of LI-DSN-Join is not critical for modern main-memory systems.

E. Impacts of Parameters

We finally study the impacts of k and ϵ by using datasets that related to the examples in Section I, i.e., Amazon, LiveJournal, Netflix, and Orkut. We omit the results of Tree, since Tree is always worse than the other algorithms.

Varying k. Fig. 5 shows the results of the experiments which study the impact of k. (We set $\epsilon = 0.001$.) Figs. 5(a)–5(d) show the average update time, and we see that the update time of LI-DSN-Join increases as k increases. This is reasonable, since as k increases, $s.\tau$ decreases, thus its verification cost increase. An interesting observation is obtained from Figs. 5(b) and 5(d). As k increases, the update time of ALL decreases. We observed that the threshold of each set often varies (decreases when $\langle s, e\rangle$ is given) in cases of small k in LiveJournal and Orkut. (Recall that many sets are verified, i.e., multiple postings lists are scanned, when thresholds decrease.) On the other hand, the thresholds are often stable when k is large. Therefore ALL incurs more costs for updating the kNN result of each set when k is smaller.

Figs. 5(e)–5(h) show the peak memory. As k increases, all the algorithms need more memory and their memory usages increase. Because all of them employ reverse kNN lists, this result is straightforward.

Varying ϵ. Fig. 6 depicts the performances of the algorithms w.r.t. ϵ. We set $k = 8$. First, ALL is sensitive to ϵ, as shown in Figs. 6(a), 6(b), and 6(d). In ALL, when $\langle s_i, e\rangle_{del}$ is given, s_i has to access $\bigcup_{e' \in s_i} I(e')$ and compute set similarities, which incurs a larger cost compared with the case where $\langle r_i, e\rangle_{ins}$ is given. Fig. 6(c) is an exception: $|I(e)|$ is large in Netflix dataset, so insertion and deletion cases incur similar costs. Meanwhile, LI-DSN-Join is robust to ϵ and keeps outperforming ALL. In terms of memory usage, we observed that ϵ has little impact, hence we omit the detail.

VI. Related work

The problem of similarity join has been receiving significant research attention [1], [2], [3], [4], [5], [15], [27], [39], [40], [41]. Since we address the problem of dynamic set kNN self-join and focus on the exact result, we review existing studies addressing exact set similarity join and similarity join in streaming setting. Approximate solutions [42] and distributed approaches [16] are beyond the scope of this paper.

Set similarity join. Many algorithms for exact set similarity join have been developed. They employ filter-and-verification framework, and the filtering mechanism has two approaches: prefix-based and partition-based filtering.

Prefix-filtering approach has been proposed in [14], and then positional-filter and suffix-filter have been devised in [2]. (Positional-filter functions only when elements in sets are pre-sorted according to a pre-defined order, thus is not employed in hash-based data structures.) A technique that removes unnecessary entries from inverted index and enhances prefix-filter has been proposed in [15]. Literature [43] proposed a prefix-filter that groups sets with the same prefix. Recently, literature [5] has further improved the performance of set similarity join by considering set relations (i.e., the idea that similar sets have similar results). This literature specifically proposed two approaches: index-level skipping and answer-level skipping. The former approach heavily relies on preprocessing data structures, so is inappropriate to environments where sets are dynamically updated. On the other hand, the latter one uses a similar technique to ours (i.e., set similarity is computed incrementally). However, there is a clear difference: we can compute $\mathsf{Jac}(s, s')$ in $O(1)$ time by using local-index, but answer-level skipping needs $O(|s\backslash s'| + |s'\backslash s|)$ time to obtain $\mathsf{Jac}(s, s')$ because all $e \in (s\backslash s') \cup (s'\backslash s)$ are scanned.

Partition-based filtering has been devised in [1], [39]. This approach divides each set into some disjoint sub-sets, and two sets have to share common sub-sets to be similar. Unfortunately, this approach incurs a significant filtering cost, thereby is outperformed by prefix-based filtering approaches [5].

Streaming similarity join. The problem of streaming similarity search has been tackled mainly in vector data scenario [40], [44], [45], [46]. Literature [44] addressed the similarity join problem in uncertain stream setting, and the uncertain objects are low-dimensional vectors whose elements have appearance probability. Multi-way join also has been addressed in streaming setting with window constraint [45]. This work however assumes a clustered machine environment (i.e., distributed processing), so is different from our assumption. Literature [40] studied the problem of streaming similarity self-join in a d-dimensional Euclidean space. The technique in [40] has been developed by optimizing the technique for Cosine similarity search. In addition, this work assumes that vectors are generated in streaming fashion, while we assume that elements are dynamically inserted and deleted. The technique hence cannot be employed in our problem. The problem of dynamic kNN join has been considered in [46], but is totally different from our work. This is because this work also assumes not sets with different sizes but multi-dimensional points with the same dimensionality.

VII. Conclusion

In many real-life applications, data can be represented as sets, e.g., e-commerce, video on-demand services, and social network services. These applications employ the set similarity self-join, as a fundamental tool, for collaborative filtering, data cleaning, and information extraction. Because these applications often update sets by adding and removing elements, they require an efficient technique to monitor the join result. Motivated by these facts, we addressed the problem of dynamic set kNN self-join for the first time. We proposed LI-DSN-Join, which is a computationally efficient algorithm. Extensive experiments using real datasets demonstrated that LI-DSN-Join can update the join result faster than algorithms that employ existing filtering techniques.

Acknowledgment. This research is partially supported by JSPS Grant-in-Aid for Scientific Research (A) Grant Number JP26240013 and 18H04095, JSPS Grant-in-Aid for Young Scientists (B) Grant Number JP16K16056, JST CREST Grant Number J181401085.

References

[1] D. Deng, G. Li, H. Wen, and J. Feng, “An efficient partition based method for exact set similarity joins,” *PVLDB*, vol. 9, no. 4, pp. 360–371, 2015.

[2] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, “Efficient similarity joins for near-duplicate detection,” *TODS*, vol. 36, no. 3, p. 15, 2011.

[3] C. Xiao, W. Wang, X. Lin, and H. Shang, “Top-k set similarity joins,” in *ICDE*, 2009, pp. 916–927.

[4] J. Wang, G. Li, and J. Feng, “Can we beat the prefix filtering?: an adaptive framework for similarity join and search,” in *SIGMOD*, 2012, pp. 85–96.

[5] X. Wang, L. Qin, X. Lin, Y. Zhang, and L. Chang, “Leveraging set relations in exact set similarity join,” *PVLDB*, vol. 10, no. 9, pp. 925–936, 2017.

[6] Y. Zhang, X. Li, J. Wang, Y. Zhang, C. Xing, and X. Yuan, “An efficient framework for exact set similarity search using tree structure indexes,” in *ICDE*, 2017, pp. 759–770.

[7] K. Subbian, C. Aggarwal, and K. Hegde, “Recommendations for streaming data,” in *CIKM*, 2016, pp. 2185–2190.

[8] D. Amagata and T. Hara, “Mining top-k co-occurrence patterns across multiple streams,” *TKDE*, vol. 29, no. 10, pp. 2249–2262, 2017.

[9] H. Kllapi, B. Harb, and C. Yu, “Near neighbor join,” in *ICDE*, 2014, pp. 1120–1131.

[10] C. A. Gomez-Uribe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation,” *TMIS*, vol. 6, no. 4, p. 13, 2016.

[11] P. Covington, J. Adams, and E. Sargin, “Deep neural networks for youtube recommendations,” in *RecSys*, 2016, pp. 191–198.

[12] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, and S. Muthukrishnan, “One trillion edges: Graph processing at facebook-scale,” *PVLDB*, vol. 8, no. 12, pp. 1804–1815, 2015.

[13] C. C. Aggarwal, “Neighborhood-based collaborative filtering,” in *Recommender Systems*, 2016, pp. 29–70.

[14] R. J. Bayardo, Y. Ma, and R. Srikant, “Scaling up all pairs similarity search,” in *WWW*, 2007, pp. 131–140.

[15] L. A. Ribeiro and T. Härder, “Generalizing prefix filtering to improve set similarity joins,” *Information Systems*, vol. 36, no. 1, pp. 62–78, 2011.

[16] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag, “Set similarity joins on mapreduce: An experimental survey,” *PVLDB*, vol. 11, no. 10, pp. 1110–1122, 2018.

[17] S. Chang, Y. Zhang, J. Tang, D. Yin, Y. Chang, M. A. Hasegawa-Johnson, and T. S. Huang, “Streaming recommender systems,” in *WWW*, 2017, pp. 381–389.

[18] Y. Huang, B. Cui, J. Jiang, K. Hong, W. Zhang, and Y. Xie, “Real-time video recommendation exploration,” in *SIGMOD*, 2016, pp. 35–46.

[19] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu, “Tencentrec: Real-time stream recommendation in practice,” in *SIGMOD*, 2015, pp. 227–238.

[20] E. Spertus, M. Sahami, and O. Buyukkokten, “Evaluating similarity measures: a large-scale study in the orkut social network,” in *KDD*, 2005, pp. 678–684.

[21] J. Noel, S. Sanner, K.-N. Tran, P. Christen, L. Xie, E. V. Bonilla, E. Abbasnejad, and N. Della Penna, “New objective functions for social collaborative filtering,” in *WWW*, 2012, pp. 859–868.

[22] B. Yang, Y. Lei, J. Liu, and W. Li, “Social collaborative filtering by trust,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 39, no. 8, pp. 1633–1647, 2017.

[23] W. Dong, C. Moses, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” in *WWW*, 2011, pp. 577–586.

[24] M. SHAN, Y. Li, B. He, and K.-L. Tan, “Accelerating dynamic graph analytics on gpus,” *PVLDB*, vol. 11, no. 1, pp. 107–120, 2018.

[25] W. Tao, M. Yu, and G. Li, “Efficient top-k simrank-based similarity join,” *PVLDB*, vol. 8, no. 3, pp. 317–328, 2014.

[26] W. Mann, N. Augsten, and P. Bouros, “An empirical evaluation of set similarity join techniques,” *PVLDB*, vol. 9, no. 9, pp. 636–647, 2016.

[27] S. Chaudhuri, V. Ganti, and R. Kaushik, “A primitive operator for similarity joins in data cleaning,” in *ICDE*, 2006, p. 5.

[28] D. Dong, Y. Tao, and G. Li, “Overlap set similarity joins with theoretical guarantees,” in *SIGMOD*, 2018, pp. 905–920.

[29] C. Teflioudi, R. Gemulla, and O. Mykytiuk, “Lemp: Fast retrieval of large entries in a matrix product,” in *SIGMOD*, 2015, pp. 107–122.

[30] T. Akiba, Y. Iwata, and Y. Yoshida, "Dynamic and historical shortest-path distance queries on large evolving networks by pruned landmark labeling," in *WWW*, 2014, pp. 237–248.

[31] Z. Shen, M. A. Cheema, X. Lin, W. Zhang, and H. Wang, "A generic framework for top-k pairs and top-k objects queries over sliding windows," *TKDE*, vol. 26, no. 6, pp. 1349–1366, 2014.

[32] S. Kapoor, "Dynamic maintenance of maxima of 2-d point sets," *SIAM J. Comput.*, vol. 29, no. 6, pp. 1858–1877, 2000.

[33] http://jmcauley.ucsd.edu/data/amazon/.

[34] http://konect.uni-koblenz.de/.

[35] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *IMC*, 2007, pp. 29–42.

[36] http://www.cs.uic.edu/liub/Netflix-KDD-Cup-2007.html.

[37] M. Theobald, J. Siddharth, and A. Paepcke, "Spotsigs: robust and efficient near duplicate detection in large web collections," in *SIGIR*, 2008, pp. 563–570.

[38] H. Li, T. N. Chan, M. L. Yiu, and N. Mamoulis, "Fexipro: Fast and exact inner product retrieval in recommender systems," in *SIGMOD*, 2017, pp. 835–850.

[39] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 918–929.

[40] G. De Francisci Morales and A. Gionis, "Streaming similarity self-join," *PVLDB*, vol. 9, no. 10, pp. 792–803, 2016.

[41] W. Mann and N. Augsten, "Pel: Position-enhanced length filter for set similarity joins." in *GvD (Foundations of Databases)*, 2014, pp. 89–94.

[42] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *WWW*, 2014, pp. 109–118.

[43] P. Bouros, S. Ge, and N. Mamoulis, "Spatio-textual similarity joins," *PVLDB*, vol. 6, no. 1, pp. 1–12, 2012.

[44] X. Lian and L. Chen, "Similarity join processing on uncertain data streams," *TKDE*, vol. 23, no. 11, pp. 1718–1734, 2011.

[45] S. Wang and E. Rundensteiner, "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing," in *EDBT*, 2009, pp. 299–310.

[46] C. Yu, R. Zhang, Y. Huang, and H. Xiong, "High-dimensional knn joins with incremental updates," *Geoinformatica*, vol. 14, no. 1, pp. 55–82, 2010.

Appendix

A. Empirical Result of Batch Update

To validate that the possible approach for batch update is essentially the same as the one that executes LI-DSN-Join for each set update, we conducted simple experiments by varying batch size b ($k = 8$ and $\epsilon = 0.001$). We measured speedup ratio against the case where batch size is 1, w.r.t. total update time. TABLE IV shows the result on Amazon and LiveJournal, and we see that the approach makes little difference in the update time.

TABLE IV
Speedup ratio

Dataset	Amazon	LiveJournal
$b = 10$	1.003	1.001
$b = 20$	1.015	1.008
$b = 50$	1.013	1.011
$b = 100$	1.018	1.016

B. Empirical Result about Jaccard similarity vs. Cosine similarity

We present the performance of LI-DSN-Join when Jaccard and Cosine similarities are used ($k = 8$ and $\epsilon = 0.001$). Fig. 7 shows the total update time to deal with all set updates for each dataset. We see that the Cosine similarity case is very similar to that of Jaccard similarity. (We also have confirmed that LI-DSN-Join keeps outperforming the other algorithms when Cosine similarity is used.)

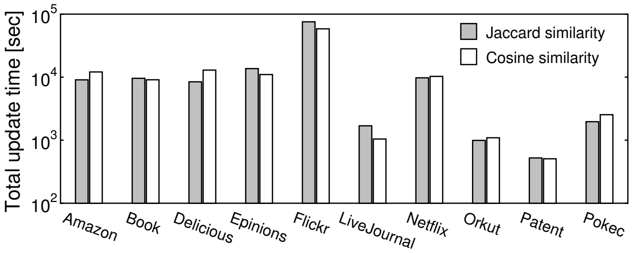


Fig. 7. Total update time when Jaccard and Cosine similarities are used