Efficient Top-K Similarity Search for Interactive Data Analysis Sessions

Technical Report

October 15, 2018

Abstract

Interactive Data Analysis (IDA) is a process in which users explore data in an incremental manner, issuing a sequence of data analysis actions (e.g. SQL queries, OLAP operations, visualizations), referred to as a session. Since IDA is a challenging process, previous work suggested the use of recommender systems to assist users in choosing the next-action to issue at each point in the session. Such recommender systems often record user sessions in a repository and utilize them to generate next-action recommendations. To do so, they search the repository for the top-k sessions most similar to the session of the current user. Clearly, the efficiency of the top-k similarity search is critical, as it allows for interactive response times.

A key observation underlying our work is that the user session progresses *incrementally*, with the top-k similarity search performed, by the recommender system, *at each step*. We devise efficient top-k algorithms that exploit this incremental nature of IDA sessions to speed up the similarity search. We first provide a simple generic model for describing IDA sessions and for capturing the commonly used, alignment-based, similarity measure for them. We then analyze the inductive properties of the alignment process and how they correlate with the incremental nature of the top-k similarity search problem, deriving novel pruning optimization techniques. Our experiments show the efficiency and effectiveness of our solution, obtaining a speedup of over 180X compared to a sequential search.

1 Introduction

Interactive Data Analysis (IDA) is fundamentally an iterative process in which a user issues a data analysis action (e.g., an SQL query, OLAP operation, visualization), receives a results set, and decides if and which action to issue next. A sequence of analysis actions issued by a user is called a *session*. IDA is a challenging process, especially for inexperienced users. Therefore, extensive research has been devoted to the development of recommender systems that assist users in choosing an appropriate next-action to perform at each point in the session.

Many of these IDA recommender systems [2,8,9,15,28] employ a *collaborative-filtering* approach and rely on a similarity comparison of user sessions. They follow the assumption that if two session *prefixes* are similar, their *continuation* is likely to also be similar. Hence, they utilize a repository of prior analysis sessions (of the same or other users): Given an *ongoing* user's session, such systems first retrieve the top-k most similar session prefixes from the repository, then examine their continuation and use the gathered information to form a possible next-action recommendation [2, 8, 9, 28].

Clearly, the efficiency of finding the top-k similar session prefixes is critical in order for IDA systems to generate recommendations in interactive response times (typically, $< 1s$). Our goal in this paper is thus to devise efficient, scalable algorithms for this top-k similarity search. Specifically, we focus our attention on a common similarity measure, originally devised and shown in [3], through a comprehensive user study, to be most suitable for the context of data analysis sessions. Consequently, it has been adopted by multiple IDA recommender systems [2, 4, 29]. The measure, which we denote by *SW-SIM* is an extension of the well-known Smith-Waterman algorithm [23] for local sequence alignment. Intuitively, given a similarity metric for the individual actions (e.g. SQL queries, OLAP, etc.), *SW-SIM* compares two sessions s, s' by *aligning* them, i.e. matching similar action pairs using an *alignment matrix*. The measure allows for (yet penalizes) gaps in the alignment, and gives a higher weight to the more recent actions in the session, since they are expected to have higher relevance on the current user's intent.

Given current user session and a sessions repository, a naive top-k search may be done sequentially by iterating over all sessions in the repository, computing the *SW-SIM* similarity score of the current session w.r.t. each of their prefixes, then selecting the top-k prefixes with the highest score. This simple algorithm, however, is prohibitively time consuming: Our experiments show that even when employed on a medium size repository of 10K sessions, **it takes the naive sequential search more than 18 seconds to complete**. To optimize it, two key challenges must be addressed:

1. A single similarity comparison of two sessions is expensive to begin with. The computation of *SW-SIM* requires the construction of an alignment matrix, which results in a high computation time (quadratic in the mean session length).

2. employing efficient indexes is highly non-trivial. Existing index structures for similarity search e.g. [5,21,24] are inadequate for *SW-SIM* since it does not induce a proper distance metric on the sessions. Also, the sessions comprise of analysis actions (which are complex objects) rather than numeric vectors, as required in e.g. [21] (See Section 2 for an elaborated discussion.)

A key observation underlying our work is that the user session progresses *incrementally* with the top-k similarity search performed by the IDA recommender system *at each step*. We exploit this property to address these two dimensions: Our first, rather direct optimization, speeds up the processing of a *single* similarity comparison between two sessions. Our second, more sophisticated optimization, tackles the *top-k search* by employing a novel threshold-based pruning technique. It employs lower and upper bounds stemming from the incremental growth of the sessions and the corresponding repetitive top-k search, accompanied by a dedicated index structure. Our experiments demonstrate that **using our optimizations a speedup of more than 200X is obtained compared to a non-optimized sequential search**.

We next provide a brief overview of our solution and contributions, then explicitly state our assumptions and focus. Related work is discussed in Section 2.

1.1 Solution Overview

Let S be a repository of sessions and let $u = q_1, q_2, \ldots$ be the ongoing user session. In the top-k similarity search problem, at each time $t = 1, 2, \ldots$ we have the prefix $u_t = q_1 \ldots q_t$ of the user session u at hand, and we need to retrieve from S the top-k session prefixes most similar to u_t. Our contributions (and the paper organization) are as follows:

Generic model for IDA sessions (Section 3) We provide a simple, generic model for representing IDA sessions, suitable for a variety of analysis platforms, from the conventional SQL/OLAP, to modern visualization tools such as Tableau and Splunk. Based on this model, we develop a formal definition of the incremental top-k similarity search problem.

Incremental-based optimizations for a single similarity comparison (Section 4) We first optimize the computation w.r.t. the user session u_t at time t and a single repository-session s. Our first immediate observation is that rather than comparing u_t to each of s's prefixes *separately*, it suffices to compute the alignment matrix of u_t and s *only once*, then derive the similarity scores of all s's prefixes simultaneously. Intuitively, this is because the matrix already includes information about the prefixes' alignment. More important, a further, more significant speedup can be achieved by reusing the computations performed at time $< t$ to enhance performance at time t. Specifically, we show that due to the same inductive nature of the alignment, the similarity scores of all s's prefixes to u_t can be simultaneously derived from scores computed in the previous iteration at time $t-1$, for u_{t-1}, refined by the new knowledge on the additional action added at time t.

Incremental-based top-k search (Section 5) We show that the information derived in the previous iterations of the top-k search can also be used to effectively prune the search space, thus avoiding an expensive sequential search. The algorithm further exploits *user idle-times* within a session to perform offline computation, thus obtaining a significant speedup. The algorithm employs two types of bounds, both derived from previous-iteration data. The first is a *global lower bound* for the similarity score of a prefix in order for it to be a member of top-k set. The second is a *per-session upper bound* on the possible similarity score of any of the sessions' prefixes to u_t. Only sessions whose upper bound is greater than the global lower bound are considered, while the rest are safely pruned (i.e., the search is exact). The two key challenges that our algorithm addresses are (1) how to derive effective bounds, and (2) how to efficiently identify candidate sessions that meet the bounds, without scanning the full repository. We explain this next.

Deriving effective similarity bounds (Section 6) Deriving exact tight lower (resp. upper) bounds is expensive since it requires full similarity scores computation. Instead, we exploit the incremental nature of the problem and use the top-k set from the previous iteration, together with the new knowledge about the last analysis action added to the user session, to derive relaxed bounds, close enough to the exact ones. We then devise a novel method for efficiently retrieving candidate sessions (who meet the similarity bounds), by analyzing the bounds and deriving from them corresponding conditions on the individual sessions' elements (i.e., their analysis actions). This allows us to use an index structure (a metric tree) for *individual actions* to quickly prune irrelevant *sessions*.

Implementation & Experiments (Section 7) We evaluate the efficiency and effectiveness of our algorithms, relative to relevant baselines, through an extensive set of experiments, on both real-life and synthetic repositories of analysis sessions, demonstrating the superiority of our solution. We use a real-life repository [1] to examine the efficiency of our algorithm in practical settings and highlight the benefits of our optimizations. Then, we use a multitude of carefully crafted synthetic repositories, each having different underlying characteristics that may affect the algorithm's performance, to demonstrate the resilience, as well as the scalability of our approach.

For space constraints, we defer all proofs to the Appendix C.

1.2 Paper Focus and Assumptions

1. Many IDA recommender systems rely on a top-k prefixes search. While a variety of tools have been developed to assist users in IDA (such as intuitive user interfaces [6] or *data driven* systems [26]), our paper is aimed at optimizing a growing number of systems [2,8,9,15,28] that rely on finding the top-k most similar session prefixes in order to generate recommendations.

2. *SW-SIM* is currently the best measure for comparing IDA sessions. Numerous definitions of similarity between two sequences exist, stemming from various application domains. [3] provides an in-depth qualitative evaluation, accompanied by a user study, determining that *SW-SIM* is most suitable for IDA sessions. Hence, we refer the reader to [3] for qualitative aspects, and focus our work, including the experimental evaluation, on performance and scalability of the top-k similarity search w.r.t. *SW-SIM*.

3. "Interactive" systems require response times of less than one second. In the domain of Human Computer Interaction (HCI), response times longer than one second are typically considered non-interactive [10]. In the context of IDA, where the user interactively examines a dataset, recommendations should be produced as fast as possible, even when utilizing a *very large repository of previous sessions*. Consequently, our goal is to support scalable top-k search in interactive speed.

2 Related Work

A variety of sequence similarity measures have been proposed in the literature in various application domains, with dedicated optimizations [11,17,20,25,27] designed for them. In particular, a large body of previous research suggests optimizations and approximations for other sequence alignment techniques, such as the original Smith-Waterman algorithm for local alignment, Dynamic

Time Warping (DTW) and string edit distance. As these measures stem from different application domains (e.g. biology, natural language processing, etc.) with different settings, their optimizations are inadequate for *SW-SIM* which is specifically designated for the context of IDA sessions. We next explain this remark in detail, relating to each type of measures. Then we discuss the only type (to our knowledge) of available alternative solutions, which includes general purpose top-k algorithms for non-metric space.

Bioinformatics-based optimizations for sequence alignment. Local alignment techniques are known to be extremely useful in the bioinformatics domain for tasks such as DNA and protein sequencing [14]. In this application domain, alignment is often performed between a query sequence and an extremely long *reference* string (e.g. a DNA genome), often from a fixed, small size alphabet (comprises, e.g., six nucleotides). Consequently, biology-driven optimizations such as [17, 27] exploit this particular property (small alphabet size) in order to index the large reference string. These solutions typically use a prefix/suffix tree or hash tables mapping small, common substrings to their location in the reference string. In contrast, in our context the "alphabet" (namely the analysis-actions space) is unbounded and, as explained in the next section, there is a predefined notion of distance between letters (analysis actions in our context). Such solutions, therefore, cannot be directly employed in our case.

Edit Distance and Dynamic Time Warping. Edit distance and DTW are popular alignment-based measures that are used in numerous application domains such as textual auto-correction, speech recognition and the comparison of time series. However, as noted in [3], these measures lack important properties for the context of IDA sessions. Classical edit-distance, which originated from the comparison of textual strings, assumes that letters are either *identical* or *mismatched*, which is inadequate for the context of IDA. This is because the alphabet, containing individual actions, is much larger and more complex. Also, the similarity of individual "letters" (actions) should be taken into consideration. DTW, on the other hand, does support the employment of a designated distance metric for the sequences' objects, yet assumes a fixed penalty of zero to gaps (insertion/deletion) [20]. Also, both edit distance and DTW do not support the important time-discount feature [3], which allows giving a higher weight to the more recent actions in the IDA session. Optimization works for these measures are typically based on specific properties of the measures' formulas or the particular application domain [11, 25].

General solutions for similarity search in a non-metric space. The sessions' similarity-space induced by *SW-SIM* does not form a proper metric, thus many optimizations that rely on the *triangle inequality* property

(e.g., [5, 16]) cannot be used here. Other solutions for non-metric spaces, e.g. [21, 24], assume that the sequences' objects are numeric vectors (e.g. as in Minkowski distance, Cosine distance), as opposed to sequences of abstract objects (analysis actions in our case). But even ignoring this, employing any such solutions in our context requires indexing each session prefix, which may be prohibitively large (typically by an order of magnitude compared to the number of sessions) and they do not exploit the incremental nature of the search.

Existing solutions for optimized top-k search that can support an arbitrary, non-metric similarity measure are Constant Shift Embedding (CSE) [19] and NM-Tree [22]. These solutions use *metrization* techniques to transform the non-metric space into a proper metric, thus their performance largely depends on the data at hand. We examine these alternative solutions in our experimental evaluation (Section 7.2) and show that our threshold-based algorithm is consistently superior, obtaining 190X shorter running times. This is due to computational inefficiency induced by the metrization when the underlying objects (IDA sessions) are closely similar.

Incremental Computation. incremental computation is used for optimization in a variety of database applications, including interactive association-rules and sequence mining [18], but the goals and settings are different from ours. To our knowledge, the only solutions that, similar to our work, exploit the incremental growth of the sequences for *similarity computations* are [12, 13], that suggest algorithms in which given textual strings A and B and their *distance-matrix*, compute the distance of Ax and B (where x is an additional character) in $O(|A| + |B|)$. We employ similar principles for our single-pair incremental alignment computation (Section 4.2), but importantly augment them with a novel threshold-based technique that allows to support top-k search in large repositories, avoiding the full repository scans.

3 Preliminaries

We start by providing the basic definitions for IDA sessions, then define the incremental top-k similarity search problem.

Interactive Analysis Sessions In the process of IDA, users investigate datasets via an interactive user interface (UI) which allows them to formulate and issue analysis actions (SQL queries, OLAP operations or visualizations) and to examine their result sets. Formally, we assume an infinite domain of analysis actions $\mathcal{Q}$ and model an analysis session as a sequence of actions $s = \langle q_1, q_2, \ldots, q_n \rangle | q_i \in \mathcal{Q}$. We use $s[i]$ to the i'th action in s (q_i) and s_i to

denote the session's *prefix* up to q_i, i.e. $s_i = \langle q_1, q_2, \ldots, q_i \rangle$. Therefore, $s = s_n$, and s_0 is the empty session. The user session is built incrementally. At time t the user issues an action q_t, then analyzes its results and decides whether to issue a next action q_{t+1} or to terminate the session. The session (prefix) u_t at time t thus consists of the actions $\langle q_1, \ldots, q_t \rangle$, where the following actions in the session, i.e., $q_{t+1}, ..q_n$ are not yet known. Given a repository S of (prior) analysis sessions, performed by the same or other users, we use $\mathcal{P}refix(S)$ to denote the set consisting of all session prefixes, i.e. $\mathcal{P}refix(S) = \{s_i \mid s \in S, 1 \leq i \leq |s|\}$.

To assist the user in choosing an appropriate next-action, IDA recommender systems search the repository to identify session prefixes $s \in \mathcal{P}refix(S)$ that are similar to u_t - the current user session - and use the gathered information to derive a next-action recommendation. To formally define sessions similarity, let us first consider the similarity of individual analysis actions, then generalize to sessions.

Individual Actions Similarity Given a *distance metric* for individual analysis actions $\Delta : \mathcal{Q} \times \mathcal{Q} \rightarrow [0, 1]$ over the actions domain, the *action similarity function* $\sigma(q_1, q_2)$ is the complement function defined by $\sigma(q_1, q_2) = 1 - \Delta(q_1, q_2)$ for all q_1, q_2 in $\mathcal{Q}$. Several distance/similarity measures for many kinds of analysis actions, e.g. for SQL and OLAP queries visualizations, and web-based analysis actions have been proposed in the literature (e.g. [3,9,15]) and our framework can employ any of them as long as the corresponding measure defines a metric space.

Session Similarity There are several ways to lift the similarity of individual elements into similarity of sequences [7] (e.g. using edit-distance, Dice coefficient, and Hausdorff distance). To assess which measure is the most suitable for comparing analysis sessions, the authors of [3] formulated desiderata for an ideal session similarity measure, based on an in-depth user study: (1) It should take the actions' *order of execution* into consideration, i.e. two sessions are similar if they contain a *similar* set of actions, performed in a *similar order*. (2) "Gaps" (i.e subsequences of non-matching actions) should be allowed yet penalized. (3) Long matching subsequences should be rewarded. (4) Recent actions are more relevant than old actions. The authors conclude that the only measure respecting all of the above mentioned desiderata is the *Smith-Waterman similarity measure* [23], a popular, generic measure for local sequence alignment, and propose an extension suitable for the context of analysis actions, denoted *SW-SIM* in our work. We next describe the measure.

		a	b	c	A	B
	0	0	0	0	0	0
A	0	0.24	0.19	0.13	0.66	0.58
B	0	0.19	0.53	0.47	0.58	1.47
A	0	0.30	0.47	0.53	1.28	1.38
B	0	0.23	0.66	0.58	1.19	2.28

(a) Session ϕ

		A	B	a	b	c
	0	0	0	0	0	0
A	0	0.48	0.43	0.37	0.30	0.23
B	0	0.43	1.07	1.00	0.93	0.85
A	0	0.59	1.00	1.43	1.35	1.26
B	0	0.52	1.32	1.35	1.88	1.78

(b) Session ψ

Figure 1: Alignment Matrices for $u_4 =$ "$ABAB$"

The similarity score of two sessions is defined recursively, on increasingly growing prefixes, with the base of the recursion being the empty prefix. At each point, the similarity of the pair of actions at the end of the prefixes is considered, and the best option, score-wise, is chosen. The two actions may either be matched, in which case an award proportional to their similarity is added to the accumulated score for their preceding prefixes, or, alternatively, one of the actions is skipped over. In this case a linear *gap penalty* $0 \leq \delta \leq 1$ is deducted from the accumulated score. To reflect the fact that the matching/skipping of older actions is less important than that of recent ones, rewards/penalties are multiplied by a decay factor $0 \leq \beta \leq 1$ with an exponent reflecting how distant the actions are from the sessions' end. More formally, the sessions similarity is defined using an *alignment matrix*.

Definition 3.1 (Alignment Matrix and Similarity Score)**.** *Given sessions s, s' of lengths n, m resp., their alignment matrix $A_{s,s'} \in \mathbb{R}^{(n+1)\times(m+1)}$ is recursively defined as follows. For $0 \leq i \leq n$, $0 \leq j \leq m$:*

$$A_{s,s'}[i,j] = \begin{cases} 0, \textit{ if } i = 0 \vee j = 0, \textit{ else:} \\ max \begin{cases} A_{s,s'}[i-1, j-1] + \sigma(s[i], s'[j])\beta^{(n-i)+(m-j)} \\ A_{s,s'}[i, j-1] - \delta\beta^{(n-i)+(m-j)} \\ A_{s,s'}[i-1, j] - \delta\beta^{(n-i)+(m-j)} \\ 0 \end{cases} \end{cases}$$

The similarity score between s and s' is defined as:

$$Sim(s, s') := A_{s,s'}[n, m]$$

The values for the decay factor β and the gap penalty δ are typically chosen by the system administrator using an extrinsic evaluation, as explained in Appendix B.1. To illustrate the definition, consider the following example.

Example 3.2. *For simplicity, assume that the action space is represented by the English letters, both uppercase and lowercase, and assume the following action similarity for any two distinct uppercase letters X,Y and their corresponding lowercase versions x,y.*

$$\sigma(\cdot,\cdot) := \begin{cases} \sigma(X,X) = 1 \\ \sigma(x,x) = 1 \\ \sigma(x,X) = \sigma(X,x) = 0.5 \\ \sigma(x,y) = \sigma(X,Y) = 0.1 \\ \sigma(X,y) = \sigma(y,X) = 0 \end{cases}$$

Identical letters are maximally similar, and two instances of the same letter, but in different case (upper/lower), are 0.5 *similar. Different letters but of the same case are* 0.1 *similar. Let* u_4 = *"ABAB" be the current user session, and consider two repository sessions* ϕ = *"abcAB" and* ψ = *"ABabc". Intuitively, according to the desiderata above, we expect* ϕ *to be more similar to* u_4 *than* ψ*, as their most recent suffix ("AB") is identical. This is reflected also in the alignment matrices of* ϕ *and* ψ*, depicted in Figure 1a and Figure 1b (resp.), when setting the gap penalty* $\delta = 0.1$ *and decay factor* $\beta = 0.9$*. The bottom-right cell in each matrix reflects the alignment score of the two sessions. The highlighted cells describes the alignment "trace", namely the cells chosen (among the three options in alignment formula) when advancing to the next step in the final score computation. Specifically, when the highlighted trace moves vertically/horizontally we have a gap, and when it moves diagonally the corresponding actions are matched. As we can see, the similarity score* $Sim(u_4, \phi)$ *is higher than* $Sim(u_4, \psi)$*.*

Our definition of *SW-SIM* follows that of [3,23] with a minor adaptation to our context. First, we define the similarity score as the *bottom-right* cell of the alignment matrix, as opposed to the *maximal* cell value in [3,23]. This is because we focus on measuring the similarity of two sessions - the user session and the repository-session *prefix* that is being examined - rather than take maximal sub-sequence similarity[1]. Second, we apply a decay factor to both actions matches and gaps, as opposed to only matches in [3], since both lose significance as the session advances. Last, we note that [3] suggests dynamically setting the decay factor and gap penalty, which are both constants in our case. As we shall see, the use of constant parameter values facilitates effective optimization via computation factorization.

Problem Definition Recall that the user session $u = \langle q_1, q_2, ... \rangle$ is built incrementally. At each step $t = 1, \ldots, |u|$ we are given u_t and wish to identify its top-k most similar session prefixes in the repository. Formally,

[1]We do so since several prefixes from the same repository-session may be of high similarity, rather than just one.

u \ ϕ		a	ab	abc	abcA	abcAB
ABAB	0	0.35	0.90	0.71	1.32	2.28
ABABc	0	0.22	0.71	1.23	1.09	1.95

(a) u, ϕ

u \ ψ		A	AB	ABa	ABab	ABabc
ABAB	0	0.80	1.81	1.67	2.09	1.78
ABABc	0	0.62	1.53	1.47	1.78	2.19

(b) u, ψ

Figure 2: Similarity vectors for u_4 and u_5

Definition 3.3 (Incremental Top-k Similarity Search)**.** *Given a user session u_t at time t and a sessions repository S, the set $top_k(u_t, S) \subseteq \mathcal{P}refix(S)$ consists of k session prefixes s.t. $\forall s \in top_k(u_t, S)$, $\forall s' \in \mathcal{P}refix(S) \setminus top_k(u_t, S)$: $Sim(u_t, s) \geq Sim(u_t, s')$. The* Incremental Top-k Similarity Search Problem *is to compute, for $t = 1, \ldots, |u|$, the set $top_k(u_t, S)$.*

For simplicity, we assume that in the course of a user session u, the repository S is unchanged. In Section 6.2.2 we discuss the minor changes required in our framework to support the case of a dynamic repository where sessions are incremented or added.

4 Incremental Similarity Computation

We first optimize the processing of a single repository session s. This will serve as an important building block in our algorithm for the incremental top-k similarity search. Given an ongoing user session u and another repository-session $s \in S$ our goal is to compute, at each time $t = 1, \ldots, |u|$ the similarity score of u_t and each of the prefixes of session s. To do so, one can naively construct the alignment matrix of u_t and every prefix of s from scratch. However, by examining the inductive construction of the alignment matrix we can derive a simple yet effective optimization, improving over the naive construction along two dimensions: (1) We show that it is sufficient to compute one alignment matrix between two sessions, then use it to simultaneously derive *all prefixes similarity scores*. (2) The alignment matrix $A_{u_t,s}$ at time t can be efficiently constructed by reusing the previous matrix $A_{u_{t-1},s}$ computed at time $t-1$.

4.1 Similarity Vectors

We explain how the alignment matrix $A_{u_t,s}$ between the current user session u_t and session s can be used to compute a *similarity vector* $\vec{Sim}(u_t, s)$, of size $|s|$, recording the scores of u_t and each of the prefixes of session s. Formally, $\vec{Sim}(u_t, s) = [Sim(u_t, s_0), Sim(u_t, s_1), \ldots, Sim(u_t, s_{|s|})]$. We use $\vec{Sim}(u_t, s)[j]$, where $j = 1 \ldots |S|$, to denote the j element in the vector. We employ this simple observation for deriving the prefixes similarity from a given alignment matrix:

Observation 4.1. *Given two session s, s' and their corresponding alignment matrix $A_{s,s'}$, the similarity score of any two session prefixes s_i, s'_j is given by:*

$$Sim(s_i, s'_j) = \frac{A_{s,s'}[i,j]}{\beta^{(|s|-i)+(|s'|-j)}}$$

From the observation above, the similarity vector $\vec{Sim}(u_t, s)$ can be computed as follows:

Proposition 4.2. *Given an alignment matrix $A_{u_t,s}$, $\forall 0 \leq j \leq |s|$: $\vec{Sim}(u_t, s)[j] = \frac{A_{u_t,s}[t,j]}{\beta^{|s|-j}}$.*

Example 4.3. *To continue with our running example, the similarity vectors $\vec{Sim}(u_4, \phi)$ and $\vec{Sim}(u_4, \psi)$ are given in the first row of the tables in Figure 2a, and Figure 2b, resp. According to Proposition 4.2, an element in the similarity vector, e.g. $\vec{Sim}(u_4, \phi)[3]$, may be computed directly from the corresponding cell in the alignment matrix of u_4 and ϕ: $\vec{Sim}(u_4, \phi)[3] = \frac{A_{u_4,\phi}[4,3]}{\beta^2} = \frac{0.58}{0.81} = 0.71$.*

4.2 Incremental Construction of Similarity Vectors

By exploiting the incremental nature of the problem, we next show how to efficiently construct a similarity vector $\vec{Sim}(u_t, s)$ at time t, from that computed at time $t-1$, thereby avoiding explicitly building the entire alignment matrix. To do so, we generalize the dynamic programming construction of the alignment matrix (Definition 3.1) to compute the full similarity vector. Vector entries at time t are then computed by reusing entries in the previous similarity vector $\vec{Sim}(u_{t-1}, s)$, computed in time $t-1$ (the colored parts of the formula in Proposition 4.4). Formally,

Proposition 4.4. *For every repository session s, user session u and time $t > 1$,*

$$\vec{Sim}(u_t, s)[j] = \begin{cases} 0, \textit{ if } j = 0, \textit{ else:} \\ max \begin{cases} \vec{Sim}(u_{t-1}, s)[j-1]\,\beta^2 + \sigma(u_t[t], s[j]) \\ \vec{Sim}(u_t, s)[j-1]\beta - \delta \\ \vec{Sim}(u_{t-1}, s)[j]\,\beta - \delta \\ 0 \end{cases} \end{cases}$$

For proof, refer to Appendix C.

Example 4.5. *Continuing with our example, assume that the user now issues a new action "c", i.e. at $t = 5$, $u_5 =$ "ABABc". The new similarity vectors $\vec{Sim}(u_5, \phi)$ and $\vec{Sim}(u_5, \psi)$, are depicted in the bottom row of Figures 2a and 2b, resp. As before, the similarity scores $Sim(u_5, \phi)$ and $Sim(u_5, \psi)$, appear in the right-most cell of the vectors. Using Theorem 4.4 we can derive each value in the new vectors from the previous corresponding similarity vectors at time $t = 4$.*

$$\vec{Sim}(u_5, \psi)[4] = max \begin{cases} \vec{Sim}(u_4, \psi)[3]\beta + \sigma(\text{“c”}, \text{“b”}) \\ \vec{Sim}(u_5, \psi)[3]\beta - \delta \\ \vec{Sim}(u_4, \psi)[4]\beta - \delta \end{cases}$$

$$= max(1.45, 1.2, 1.78) = 1.78$$

Time Complexity Analysis Let $|\widehat{s}|$ be the average session size and λ the complexity of computing similarity for individual actions. The expected time complexity of computing the similarity vector $\vec{Sim}(u_t, s)$ using the naive approach is $O(|\widehat{s}|^3\lambda)$, since $|\widehat{s}|$ alignment matrices are constructed. Employing Proposition 4.2 allows us to compute the similarity vector $\vec{Sim}(u_t, s)$ by constructing only one alignment matrix, thus the expected complexity is reduced to $O(|\widehat{s}|^2\lambda)$. Further employing the incremental construction in Proposition 4.4 reduces the expected time to $O(|\widehat{s}|\lambda)$ since only $|s|$ action similarity calculations are required, between $u_t[t]$ (the newly added action in u_t) and all the actions in s.

5 Incremental Top-k Search

To solve the top-k similarity search problem one needs to retrieve the set $top_k(u_t, S)$ of similar prefixes, iteratively at each time $t = 1, \ldots, |u|$. The optimizations described above can be directly lifted into a top-k algorithm, denoted I-TopK, that iterates over all sessions in the repository S. However, such a full repository scan may be excessively expensive for large repositories. To avoid this, we design a novel, threshold-based algorithm, denoted T-TopK that effectively prunes the search space, while deferring some of the computation to system idle-times. To set the ground, we first briefly sketch the simpler I-TopK algorithm, then describe the optimized T-TopK algorithm.

5.1 Iterative Top-k Algorithm

The I-TopK Algorithm is depicted in Algorithm 1. It uses a repository named *vecRepo* of similarity vectors computed in previous iterations. In the algo-

Algorithm 1 I-TopK($u_t, S, k, vecRepo$)

Input: u_t, S, k, $vecRepo$.
Output: $top_k(u_t, S)$

1: $top \leftarrow MaxHeap(k)$
2: **for** session $s \in S$ **do**
3: $\quad \vec{Sim}(u_t, s) \leftarrow ComputeVector(vecRepo, u_t, s)$
4: $\quad$ **for** $(j = 1; j \leq |s|; j++)$ **do**
5: $\quad\quad top.push(\vec{Sim}(u_t, s)[j], s_j)$
6: return top

rithm we use $vecRepo[u_{t'}, s]$ to denote the stored similarity vector of $u_{t'}$, $t' < t$, w.r.t. session s. I-TopK takes as input the current session u_t, the sessions repository S, and the desired size k of the top-k set. First, an empty k-size max heap (called top) is initialized. The heap is used to store the current top-k most similar prefixes to u_t at each point of the computation (Line 1). Next, we iterate over the sessions repository (Lines 2-5). For each $s \in S$, the similarity vector $\vec{Sim}(u_t, s)$ is computed by calling the *ComputeVector* function (Line 3). Then, the scores of s's prefixes are pushed into the max-heap with the corresponding prefix (Lines 4-5). Last, the prefixes in the max-heap are returned as $top_k(u_t, S)$ (Line 6).

The function *ComputeVector*, depicted in Algorithm 2, computes $\vec{Sim}(u_t, s)$ directly from $\vec{Sim}(u_{t-1}, s)$ (as recorded in $vecRepo[u_{t-1}, s]$) following Proposition 4.4.

Note that since the computation of the similarity vectors at time t requires only the vector computed at $t-1$, it is enough to store in $vecRepo$ a single (the most recent) similarity vector for each session $s \in S$. Previous vectors are discarded, therefore the I-TopK algorithm requires $O(|S||\widehat{s}|)$ space, where $|\widehat{s}|$ is the average session size. Regarding the time complexity of the algorithm, recall that incrementally computing a single similarity vector takes $O(|\widehat{s}|\lambda)$, therefore executing the I-TopK Algorithm takes $O(|S||\widehat{s}|\lambda)$ as it iterates over the session repository S. However, a disadvantage of this algorithm is that it requires a full scan of the session repository. We next present a novel threshold-based optimization technique, augmented with efficient indexing mechanism, that allows to efficiently prune "unpromising" session candidates (i.e. with low similarity to u_t).

5.2 Threshold-Based Algorithm

The algorithm consists of six key components.

Algorithm 2 $ComputeVector(vecRepo, u_t, s)$

Input: $vecRepo$, u_t, s.
Output: $\vec{Sim}(u_t, s)$.

1: $\vec{Sim}(u_{t-1}, s) \leftarrow vecRepo[u_{t-1}, s]$
2: Initialize $\vec{Sim}(u_t, s)$, a vector size ($|s|+1$) of zeros.
3: **for** $(j = 1; j \leq |s|; j++)$ **do**
4: Compute $\vec{Sim}(u_t, s)[j]$ as in Proposition 4.4
5: $vecRepo[u_t, s] \leftarrow \vec{Sim}(u_t, s)$
6: Return $\vec{Sim}(u_t, s)$

1. Forming a global similarity threshold We first compute an initial lower bound threshold for the similarity score of a prefix to be a member of $top_k(u_t, S)$. We denote this threshold inf^t. Since the exact, *tight* threshold can only be computed after the top-k set is known (i.e., the minimum similarity score in $top_k(u_t, S)$), we use a relaxed inf^t threshold, computed a-priori, which is smaller yet close enough to this minimum.

2. Computing a similarity upper-bound for each session Next, for each session $s \in S$, we compute an upper bound threshold on the possible similarity score of any of its prefixes and u_t. We denote the computed threshold $sup^t(s)$. As in the case of the lower bound, deriving the exact upper bound requires computing the actual similarity scores for each prefix of s and take the maximum. Therefore, we devise an upper bound $sup^t(s)$ that is greater, but close enough to this maximum.

3. Pruning unpromising sessions Using the similarity lower- and upper-bounds described above, we can *prune the search space* and consider only repository sessions s whose upper bound $sup^t(s)$ is greater than the current lower bound inf^t, as these are the only sessions that have the potential to enter the top-k set. We call these the *candidate sessions*. It is important to note that since our computed lower (upper) bounds are smaller (greater) than the exact bounds, our algorithm does not miss any valid top-k candidates.[2]

4. Computing similarity vectors For each candidate session s we compute its similarity vector $\vec{Sim}(u_t, s)$ and, as in the previous I-TopK algorithm, push the similarity scores of s's prefixes into the max-heap. Recall however that to efficiently compute the similarity vector at time t, we need to have

[2]Naturally, redundant sessions may be examined, but as we show in the experiments, not too many.

the vector of time $t-1$ available in the repository. While this was guaranteed in the I-TopK algorithm, it is no longer the case here, since due to pruning, s might have been skipped in previous iterations. Thus, the missing similarity vectors must be completed before $\vec{Sim}(u_t, s)$ can be computed.

5. Offline computation in user idle-times User idle-times occur when the user examines the results of her current action and before deciding on the next one (which typically takes serveral seconds). Our threshold-based algorithm allows utilizing such idle-times, to compute as many missing vectors as possible *offline*, so that they are already available when needed in the top-k search.

6. Refining the bounds As we compute more similarity vectors of candidate sessions, and accordingly update the max-heap, the minimum similarity score required to enter the top-k set monotonically increases. Therefore we can further refine the candidates set by pruning any session s whose upper bound is lower than the current minimum score in the heap, i.e. where $sup^t(s) < min-score(top)$.

Algorithm 3 depicts the pseudo-code of our threshold-based framework, denoted T-TopK. For a given user session u_t, a sessions repository S and a number k, the prefixes set $top_k(u_t, S)$ is retrieved as follows: First, if u_t contains only one action, the I-TopK algorithm will be used, since we have no previous prefix to rely on. Otherwise, we compute the lower-bound similarity threshold inf^t (Line 5), and use it to find the candidate sessions having similarity upper-bounds greater than inf^t (Line 6). For each such candidate session we complete missing vectors in vecRepo, if necessary (Lines 8- 9), then compute the similarity vector (Line 10). Next, we iteratively push each value of the similarity vector (each representing a prefix similarity score) into max heap *top* of size k (Lines 11- 12), and further prune candidate sessions if their upper bound is lower than the minimum score in *top* (Lines 13-14). Finally, the max-heap *top* holds the set $top_k(u_t, S)$, containing the top-k most similar prefixes to u_t.

The offline procedure works as follows: It is invoked after a user issues an action and runs until she issues the next one. During this time it repeatedly pulls repository sessions s whose similarity vector has not been computed in the previous top-k computation, then computes their missing vectors, and inserts them to the vectors repository *vecRepo*.

In Appendix C we prove the correctness of T-TopK, i.e., that it always returns the exact set $top_k(u_t, S)$.

To complete the picture we still need to explain how (1) the similarity

Algorithm 3 T-TopK($u_t, S, k, vecRepo$)

Input: $u_t, S, k, vecRepo$
Output: $top_k(u_t, S)$

1: $top \leftarrow MaxHeap(k)$
2: **if** t == 1 **then**
3: Return $I - TopK(vecRepo, u_t, S, k)$
4: **else**
5: Compute inf^t, the lower-bound similarity threshold
6: $C \leftarrow \{s | s \in S \wedge sup^t(s) \geq inf^t\}$
7: **for** session $s \in C$ **do**
8: **for** $(i = max\{t' | \vec{Sim}(u_{t'}, s) \in vecRepo\} + 1; i < t; i++)$ **do**
9: $computeVector(vecRepo, u_i, s)$
10: $\vec{Sim}(u_t, s) \leftarrow computeVector(vecRepo, u_t, s)$
11: **for** $(j = 1; j \leq \lfloor s \rfloor; j++)$ **do**
12: $top.push(\vec{Sim}(u_t, s)[j], s_j)$
13: **if** $|top| == k$ **then**
14: $C \leftarrow C \setminus \{s | sup^t(s) < min-score(top)\}$
15: return top

lower bound inf^t, and (2) the upper bounds $sup^t(s)$ are computed and compared. But before we do so, let us first abstractly analyze the complexity of the T-TopK algorithm, so that we know what to require from this computation.

Algorithm Analysis Let $|\widehat{C}|$ be the expected number of candidate sessions meeting the threshold in Line 6 (We assume $|\widehat{C}| >> k$) and let $\widehat{\tau}$ be the expected number of missing similarity vectors that need to be computed in Lines 8-10 for each candidate session (i.e., the mean $t - max\{t' | \vec{Sim}(u_{t'}, s) \in vecRepo\}$). Also, denote by α the complexity of forming the similarity thresholds and computing the candidates set. Thus the average time complexity of T-TopK is given by $O(\alpha + \lambda|\widehat{C}|\widehat{\tau}|\widehat{s}|)$ (We discuss the space requirements in the next section). Recall that the average complexity of the simple I-TopK algorithm is $O(\lambda|S||\widehat{s}|)$. The answer to the question "Which algorithm is preferable?" in terms of running times naturally depends on how small α, $|\widehat{C}|$ and $\widehat{\tau}$ are.

6 Lower and upper bounds

We first explain how the similarity lower and upper bounds are computed, then show an efficient procedure to retrieve candidate sessions whose upper bounds are greater than the lower bound.

6.1 Lower Bound Similarity Threshold

Our goal is to derive a lower bound inf^t for the similarity score of a prefix to be a member of $top_k(u_t, S)$ which is close enough to the exact bound yet can be computed efficiently.

Intuitively, we use the sessions in the (already computed) top-k set of time $t-1$ as a potential representative for the top-k set of time t, and define our threshold w.r.t. them. Let $S^{t-1}_{top} \subseteq S$ denote the set of sessions with prefixes in $top_k(u_{t-1}, S)$. Rather than computing the top-k set from the entire repository S we compute it from (the much smaller) S^{t-1}_{top}, and define our threshold as:

$$inf^t = min\text{-}score(top_k(u_t, S^{t-1}_{top}))$$

Note that since the size of S^{t-1}_{top} is at most k, inf^t is computed efficiently. The following proposition shows that inf^t is indeed a lower bound threshold. It also tells us how far, at most, it may be from the tight bound.

Proposition 6.1. $min\text{–}score(top_k(u_{t-1}, S))\beta - \delta \leq inf^t \leq min\text{–}score(top_k(u_t, S))$

For illustration, consider the following example:

Example 6.2. *Assume that $k = 1$ (i.e. we are interested in the top-1 similar prefixes), and consider the sessions u, ϕ, ψ from our running example. Considering the similarity vectors at $t = 4$ (detailed in Example 4.3), the most similar prefix is ϕ_5 and thus $top_1(u_4, \{\phi, \psi\}) = \{\phi_5\}$. Consequently, its corresponding session is in the set $S^4_{top} = \{\phi\}$. For computing the threshold $inf^5 = min - score(top_1(u_5, S^4_{top})$, we construct the similarity vector $\vec{Sim}(u_5, \phi)$ (bottom row in Figure 2a), and take the maximal similarity score among its elements, thus $inf^5 = 1.95$.*

6.2 Upper-Bounding the Similarity Scores

For each session $s \in S$, we want to upper bound the possible similarity score of its prefixes to u_t.

Let $\widehat{sup}^t(s) = max_{1 \leq i \leq |s|} Sim(u_t, s_j)$ denote the *tight*, exact upper bound for s.

Recall that for sessions in S_{top}^{t-1}, we already computed their similarity vectors when computed the lower bound inf^t, therefore their tight bound is already at hand. Nevertheless, for the rest of the sessions, i.e. $s \in S \setminus S_{top}^{t-1}$, we show that the incremental nature of the computation can be used to form $sup^t(s)$, a looser yet effective *proper* upper bound to $\widehat{sup}^t(s)$.

Definition 6.3 (Proper upper bound)**.** *We say that $sup^t(s)$ is a* proper upper bound *(w.r.t u_t and S), if for every session s having some prefix s_j where $Sim(u_t, s_j) > inf^t$, we have that $sup^t(s) \geq \widehat{sup}^t(s)$.*

Observe that *proper* upper bounds are not always strict upper bounds, in the sense that for some sessions, namely those where all their prefix similarities are below threshold, $sup^t(s)$ may not be greater than $\widehat{sup}^t(s)$. Nevertheless, We can show that Algorithm 3 is sound and complete when $sup^t(s)$ is a proper upper bound. This follows from the fact that if all prefixes of a session s have similarity scores lower than inf^t, then none of them can be in the top-k set.

To complete the presentation we first (1) state our proper upper bounds then (2) show how candidate sessions that satisfy the bounds can be efficiently identified.

6.2.1 Upper Bounds

Our goal is to be able to quickly identify sessions having some prefix whose similarity to u_t is above the threshold inf^t while pruning the rest of the sessions.

We show that unless the last action in u_t, denoted q_t, is aligned to an action in $s \in S \setminus S_{top}^{t-1}$, it can be safely pruned.

Proposition 6.4. *For a session $s \in S \setminus S_{top}^{t-1}$, if when computing $Sim(u_t, s)$ the action q_t is not matched with any of s's actions, then for every prefix s_j of s, $Sim(u_t, s_j) \leq inf^t$.*

The proof (See Appendix C) is based on the observation that $min-score(top_k(u_{t-1}, S))\beta - \delta$ is a lower bound on inf^t. Consequently, in the analysis below we ignore sessions where q_t is not matched with any of s's actions, and for brevity, unless stated otherwise, whenever we refer to a session s we mean one in $S \setminus S_{top}^{t-1}$ where q_t had a match.

As we show next, for a given session s, there are two intuitive ways to bound $\widehat{sup}^t(s)$ from above *without actually computing the prefixes similarity.* $sup^t(s)$ is then defined as the minimum value among these two bounds. In our implementation (to be detailed in the next subsection) we use the first bound to quickly prune irrelevant sessions. The full bound is then computed only for the selected sessions, to further prune the candidates set.

First bound (B_1) The following proposition shows that we can bound $\widehat{sup}^t(s)$ from above using the top-k set of the previous iteration and the maximal similarity of the session's individual actions to the new action q_t.

Proposition 6.5. *For every session s, $\widehat{sup}^t(s) \leq min\text{–}score(top_k(u_{t-1}, S))\beta^2 + max_{q \in s}\{\sigma(q_t, q)\}$.*

Intuitively, this is because for every prefix that is not at the $top_k(u_{t-1}, S)$, the similarity to u_{t-1} is smaller than $min\text{–}score(top_k(u_{t-1}, S))$. Consequently, even if the newly added action q_t is matched to the most similar action in s, by the definition of the similarity measure, the cumulative score cannot be greater than the prefix similarity (multiplied by the decay factor, following the arrival of a new action) plus the similarity of the best match. Consequently, we use the following as our first bound:

$$B_1(s) = min-score(top_k(u_{t-1}, S))\beta^2 + max_{q \in s}\{\sigma(q_t, q)\}$$

Note that since $top_k(u_{t-1}, S)$ has already been computed, to calculate the bound we only need to compute the similarity of q_t and the actions in s. As the actions reside in a metric space, we show in Section 6.2.2 how this is done efficiently.

Second bound (B_2) We devise an alternative way to bound $\widehat{sup}^t(s)$, without computing the actual prefixes' similarity to u_t. For space constraints we briefly describe below the main idea and refer the reader to Appendix A for further details, example and proof. Intuitively, the alternative bound for $\widehat{sup}^t(s)$ is achieved by dividing u_t into three disjoint subsequences w.r.t. a session s, and separately bounding the similarity to each part: The segment of u_t that was already compared to s in previous iterations, the segment containing only the last added action q_t, and the segment in between these two. By adding up the bounds for each segment we obtain a bound for the whole sequence:

Proposition 6.6.

$$\widehat{sup}^t(s) \leq max_{1 \leq j \leq |s|}\{\vec{sim}_{u_\tau, s}[j]\}\beta^{2(t-\tau)} + \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1} + \\ + max_{q \in s}\{\sigma(u_t[t], q)\}$$

Note that bound B_2 (the right-hand side expression of the inequality above) requires no action comparisons besides those already performed for B_1, and that all other values are either predefined constants or already computed in previous iterations. The proper upper bound $sup^t(s)$ is defined as the minimum of the two bounds: $sup^t(s) = min(B_1(s), B_2(s))$

6.2.2 Efficient Retrieval of Candidate Sessions

As mentioned above, we use the first bound B_1 to quickly identify relevant sessions, then compute the full bound only for the retrieved sessions, for further pruning. We use the following observation, which follows immediately from Proposition 6.5.

Observation 6.7. *For $sup^t(s)$ to be not smaller than inf^t, s must contain some action q s.t. $\sigma(q_t, q) \geq inf^t - min-score(top_k(u_{t-1}, S))\beta^2$*

To identify sessions that contain such actions, we employ an index structure that uses the fact that the action similarity measure defines a *metric space* (See Section 3). Specifically, in our implementation we use a *metric-tree* [5] - a popular index structure that harnesses the triangle inequality property of a metric space to facilitate a fast similarity search.

Sessions selection via the actions metric-tree We maintain a metric-tree to index the sessions' actions, with pointers from each action to the session it appears in.[3]. The distance function used is the complement of our predefined action similarity function, i.e. $\Delta(q, q') = 1 - \sigma(q, q')$.

From Observation 6.7 and our definition of the action distance notion, it follows that all sessions that satisfy the bound B_1 must contain some action q s.t. $\Delta(q_t, q) \leq 1 - inf^t + min-score(top_k(u_{t-1}, S))\beta^2$. We thus use the metric tree for a fast retrieval of all such actions q, and follow their associated pointers to identify the relevant sessions. Now we only need to compute the full upper bound for the retrieved sessions to select those with upper bound greater than inf^t.

The following example illustrates the operation of the efficient candidates search using the metric tree.

Example 6.8. *For our running example, at $t = 5$ we retrieve candidate sessions via the metric tree, w.r.t. the lower bound* 1.95 *(computed as in Example 6.2). Recall that our goal here is to retrieve sessions having an action q s.t. $\Delta(q_5, q) \leq 1 - inf^5 + min-score(top_1(u_4, \{\phi, \psi\}))\beta^2$, namely actions similar to q_5 = "c" with distance no more than $1 - 1.95 + 2.28 * 0.81 = 0.897$. Note that only the letters "C" and "c" meet this constraint, therefore sessions ψ (that contains "c") is retrieved and added to the initial candidate sessions set. Then, the upper bound for ψ is given by $sup^5(\psi) = min(\{B_1(\psi), B_2(\psi)\}) = min(2.84, 3.09) = 2.84$ (full computation omitted), which is greater than the lower bound* 1.95*, therefore final set of candidate sessions is $\{\psi\}$.*

[3]Syntactically-identical actions from different sessions are stored separately.

Additional Remarks We conclude this section with two remarks. First, regarding the space complexity, on top of storing the similarity vectors there is an additional cost induced by maintaining the metric tree. However, assuming that the metric tree is balanced, the overall space complexity of the algorithm is thus $O(|S||\widehat{s}|)$, which is the same as for I-TopK. In Section 7 we show that the additional cost induced by the metric tree is marginal, even when the session repository contains over $1.5M$ individual actions.

Second, we note that in practical settings the sessions repository is dynamic, i.e. new sessions are added and existing ones are incremented. T-TopK can be easily adapted to this setting, as newly added or incremented repository sessions merely induce more *missing similarity vectors*. Recall from Section 5.2 that missing vectors can be computed during *user idle-times* (between her consecutive actions) thus have minimal effect on the computation time of the top-k search.

7 Experimental Study

We conducted extensive experimental evaluations on both synthetic and real-life session repositories. Since the only publicly available repository (to our knowledge) of real-life analysis sessions is rather small [1], we first examined the resilience and scalability of our approach on a multitude of synthetic repositories, and used the small real-life repository to verify that the same performance trends occur on it as well.

Last, let us recall that our experiments focus on performance only, and we refer the reader to [3] for the comparative qualitative study of *SW-SIM*.

7.1 Experimental Datasets & Setup

We first describe the repositories we used, then explain the algorithms' implementations.

Synthetic Session Repositories We constructed a multitude of carefully crafted, synthetic repositories. Each repository has different characteristics, such as the level of similarity between actions and sessions, and scale-effecting parameters - the repository size and the (mean) session length. For brevity, we only provide a high-level description of the construction routine, and refer the reader to Appendix B.4 for the complete details. In short, to generate a synthetic repository, we first built an action metric-space with varying similarity among the actions, then built each repository session by drawing individual actions from this space, varying the sessions' resemblance. Actions

Parameter	Min	Max	Default
Problem Parameters			
Decay Factor β	0.1	1	0.9
Gap Penalty δ	0.05	1	0.1
Output Size k	4	24	12
Controlled Repository Parameters			
Action Dim.	5	500	25
#Action Clusters	3K	24K	6K
Cluster Rad. (std)	0.0075	0.012	0.003
#Seed Sessions	6%	16%	10%
%Action Pairs (p)	0%	100%	80%
Idle Time (s)	0	5	1
Repository Scale Parameters			
#Sessions	1K	100K	10K
Session len.	$\mathcal{N}(4, 3^2)$	$\mathcal{N}(32, 12^2)$	$\mathcal{N}(16, 3^2)$

Table 1: Problem & Repository Parameters

are abstractly modeled as points in a multi-dimensional space. The controlled parameters, as depicted in Table 1 are as follows: *Action dimensions* stands for the number of dimensions of the vector that represents a single action. *#Action Clusters* and *Cluster Radius* control the number of the inherent action-clusters and their distance radius (standard deviation) in the action metric-space. The parameters that control the session repository are *#Seed Sessions* and *Action Pairs (p)* which stand for the number of "prototype" sessions that serve the basis for constructing the rest of the session, and the percentage of actions in each repository session that is from the same clusters as the actions of its corresponding seed. Finally, *Idle-Time* corresponds to the mean user idle-time between two consecutive user actions. Using this construction method, we created a multitude of synthetic repositories, each with a different parameters setting.

REACT-IDA Session Repository We used the only publicly available repository (to our knowledge) of real-life sessions [1], collected as part of the experimental evaluation of an existing IDA recommender system [15] (developed by a subset of the authors of this work). The user sessions were performed by 56 analysts who used REACT-UI, a web-based analysis interface, in order to explore four different datasets from the domain of cyber-security. The repository contains a total of 1100 distinct actions. The average session length in this repository is 8.5 actions, and the median user idle-time is 40 seconds.

Prototype Implementation We implemented the algorithms presented in the previous sections in *Java 8*, using Guava (`https://github.com/google/guava`) for the max-heap. For the metric-tree we used an M-tree [5] Java implementation (`https://github.com/erdavila/M-Tree`), employing *Minimum Sum of Radii* split policy (See [5]), and a maximum node capacity of 20 objects. All experiments were conducted over *Intel Core i7-4790, 3.6GHz* machine (4 dual cores), equipped with 8GB RAM and running *Windows 7*. Recall that both algorithms require a measure for action similarity. When using the real-life repository, we employed the action similarity measure of the IDA recommender system [15] whose analysis UI was used to record the sessions.[4] For the synthetic repositories, where the actions are represented as points in a multidimensional space (See Appendix B.4) we used their Euclidean distance to calculate the actions' similarity.

7.2 Baseline Comparison

We compared the performance of the T-TopK algorithm to the following baseline algorithms, each computing the exact set $top_k(u_t, S)$ according to the alignment based similarity measure (Definition 3.1). Among these, we show the results of the following baselines: (1) **Naive Sequential Search (NSS)** that retrieves the set of similar prefixes by iteratively comparing u_t to each prefix of each repository session in S using a direct implementation of Definition 3.1 with no further optimizations, then selects the top-k similar prefixes. (2) **Optimized Sequential Search (OSS)**, which employs the optimization in Section 4.1, instead of computing the alignment matrix for each prefix. Namely, it iterates over all sessions in $s \in S$, constructs a single alignment matrix $A_{u_t,s}$, and uses Observation 4.1 to derive the similarity scores of u_t and each prefix of s. (3) **I-TopK**, the iterative algorithm depicted in Section 5.1 which employs both the optimizations in Section 4.1 and 4.2. (4) **T-TopK**, the threshold based algorithm as depicted in Sections 5.2 and 6. (5) **Constant Shift Embedding (CSE) [19]** and (6) **NM-Tree [22]**, are general purpose solutions for top-k search in a non-metric space. Both use metrization techniques: CSE use a simple solution that increments each distance score by a predefined constant, so the triangle inequality is enforced. Then, all session prefixes in S are stored in a *metric tree* (w.r.t. the new metric space). When given the ongoing session u_t, the metric tree is traversed, using the new metric to obtain the exact set of top-k similar prefixes. NM-tree uses a more sophisticated similarity-preserving transformation on the

[4]The similarity function takes into consideration both the action syntax and (a signature of) the results. For details see [15].

Baseline	REACT-IDA	Repo-10	Repo-100
	Time (ms) / #ops	**Time (ms) / #ops**	**Time (s) / #ops**
NM-Tree	42.3 / 27.5K	18,503 / 12.3M	186 / 123M
CSR	55 / 36.6K	18,224 / 12.1M	183 / 121M
NSS	51.9 / 35K	17,204 / 11.5M	170 / 123M
OSS	7.6 / 5.1K	3,242 / 1.36M	32 / 13.6M
I-TopK	1 / 1.1K	237 / 160K	2.4 / 1.6M
T-TopK	**0.5 / 641**	**59 / 39K**	**0.9 / 722K**

Table 2: Baselines Results

original (non-metric) distance measure, then employs a dedicated extension of the metric tree to index and query the transformed metric space.

Finally, recall from Section 2 (Related Work) that optimization techniques dedicated to other similarity measures cannot be directly utilized on *SW-SIM*, therefore could not serve as additional baselines.

Evaluation Process We evaluated all baselines using a multitude of configurations for the top-k problem (i.e. the constants β, δ of the similarity measure, and k) and for the synthetic repositories. Table 1 presents the value ranges of the parameters used to construct the different configurations. The default problem parameters were determined by an extrinsic evaluation, following those of the IDA recommender system in [15] (See Appendix B.1). The default synthetic repositories configuration uses intermediate values.

The evaluation process is as follows. Given a session repository S, in each trial we draw a random session prefix as u_t, then employ each baseline to retrieve the set $top_k(u_t, S)$ of similar session prefixes. We performed 100 trials for each repository, capturing the average execution time and memory consumption, as well as the average number of action similarity operations performed in each run.

Results Table 2 presents a representative sample of the results, showing the average running time and the number of query similarity operations (denoted #ops) obtained by each baseline compared on the REACT-IDA repository (Containing 1.1K actions) as well as Repo-10 and Repo-100, which are two synthetic repositories in their default configurations (See Table 1) containing 10K and 100K sessions (resp., 160K and 1.6M actions).

First, for both Repo-10 and Repo-100, the performance of *NM-Tree* and *CSR* is almost on-par with NSS (the naive sequential search). This could be due to the fact that transforming the non-metric space may induce high overlap between the metric-tree nodes, therefore the search deteriorates to sequential search plus additional overhead induced by the metric tree (See [22]).

Second, note that OSS improves running times by $5X$ (and #ops by $9X$) over NSS due to its efficient use of Observation 4.1. However, compared to OSS, a more significant improvement of $15X$ in running times (and $9X$ in terms of #ops) is achieved by I-TopK which utilizes the incremental computation. Finally, additional 2-4X speedup is obtained by T-TopK, **resulting in an overall improvement between 207X to 294X in running times, over the NSS baseline**.

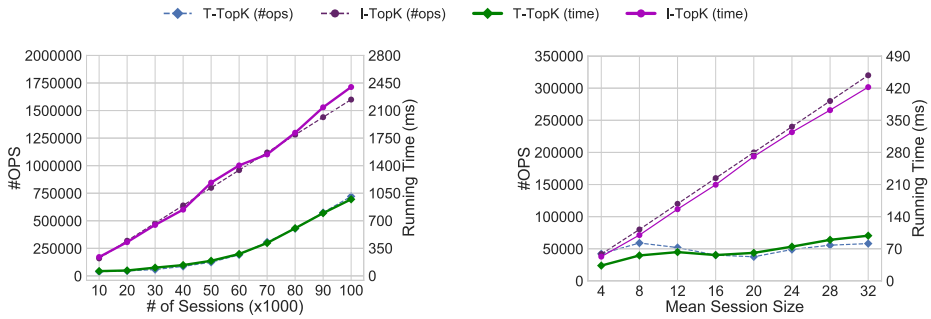
The performance trends on the REACT-IDA repository (comprising the smaller collection of real-life sessions) are similar, with some minor variation. We can first see that NM-Tree performs slightly better than NSS and CSR. This is due to the specific underlying distribution of similarity scores in the dataset, which is slightly more favorable here for the NM-Tree transformation. But here too our algorithms perform significantly better, with T-TopK dominating. This is interesting since, while one can expect that the pruning-based optimizations of T-TopK will be less effective on a small repository, it retains a $2X$ speedup compared to I-TopK, and significantly outperforms the other baselines.

Next, we present an in-depth performance comparison of the algorithms when employed on a multitude of different repositories and with different problem parameters, as well as a deeper examination of the computation segments in T-TopK. Since in all configurations, the performance of T-TopK was significantly better (by at least an order of magnitude) than the rest of the algorithms, we omit them from presentation and use only T-TopK as a baseline.

7.3 Scalability & Parameters Effect

We next analyze the performance of the T-TopK Algorithm compared to I-TopK, by varying the problem parameters as well as the repository parameters, one at a time, while keeping the rest at their default values (As in Table 1). We measured both the running times, number of action similarity operations (*#ops*), and memory consumption.

Scalability Parameters Figure 3a depicts the performance of the algorithms when increasing the number of sessions in the repository. For both algorithms we can see a rather linear increase in running times (correspondingly, *#ops* however the two plots overlap), but T-TopK shows a significant speed up of 3X (on average) over I-TopK. Moreover, as we can see in Figure 3b, when the mean *session length* increases, T-TopK performance stays stable compared to a linear increase for I-TopK. This demonstrates that the



(a) Repository Size (b) Mean Session len.

Figure 3: Effect of Scale Parameters

effectiveness of the threshold-based approach further grows for longer sessions. As for memory consumption, the maximal usage for T-TopK did not exceed $74MB$ even for a repository as big as 100K sessions (1.6M individual actions), which is negligible in practice, so we do not further discuss in the next experiments.

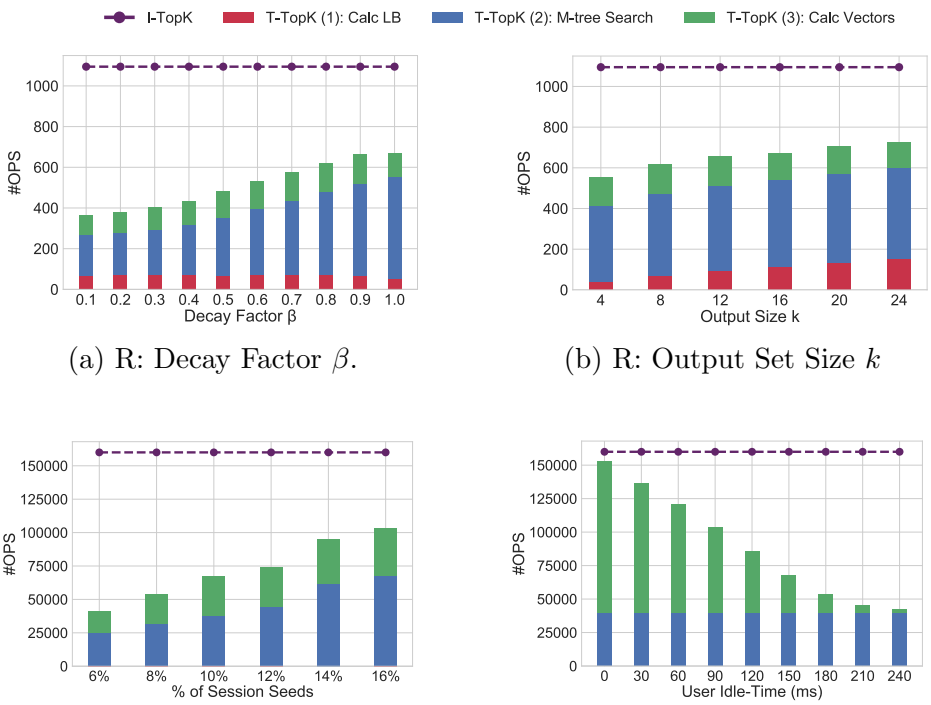
We next present a high-level overview of the effect on performance induced by varying the problem and the repository parameters. For a detailed examination of each parameter, refer to Appendix B.4.

Effect of Problem Parameters Recall that the problem parameters are the gap penalty δ, the decay factor β and the output set size k. We examine their effect on the performance, when varying each parameter, and keeping the rest at their default values mentioned in Table 1. To gain a better insight on the computation of T-TopK, we break the total $\#ops$ performed into three computational segments: (1) forming the initial similarity lower bound, (2) the metric-tree search, and (3) computing similarity vectors for candidate sessions. The corresponding running time trends are similar, therefore omitted from the figures below. Also, as Segment (3) is negligibly small when using the default idle-time, we further restrict the default user idle-time to 150ms[5], which naturally stresses T-TopK.

The performance of I-TopK in terms of #ops is represented in the figures by a dotted flat line since it is not dependent on the problem parameters.

In general, increasing each of the problem parameters resulted in a minor increase in #ops for T-TopK. Figure 4a and Figure 4b show, for example, the effect of the decay factor β and the output size k on performance, when the algorithms are employed on the REACT-IDA repository. The increase in performance is expected, since the values of the problem parameters play a

[5]For comparison, the minimal human reaction time to a stimulus is 0.2 seconds.



(a) R: Decay Factor β.

(b) R: Output Set Size k

(c) S: % Seed Sessions

(d) S: Idle-Time

Figure 4: Parameters Effect on REACT-IDA (R) and Synthetic (S) Datasets

part in the lower/upper bound computations: (1) The decay factor β affects the upper bound, therefore lower values induce more restrictive candidate selection, and thereby better performance. (2) Increasing the output size k causes a decrease in the lower bound threshold inf^t, thus more candidate sessions are examined.

Last, we note that similar trends were observed for varying the gap penalty δ, as well as for all parameters on the synthetic repositories (See Appendix B.3 for more details).

Repository Parameters Repository parameters affect the likelihood of two arbitrary actions/sessions to be similar. The higher the similarity is, the better the performance of T-TopK compared to I-TopK. For instance, Figure 4c depicts performance for varying number of seed sessions (i.e., session "cluster centers"), from 6% to 16% of the sessions repository size. As expected, increasing the number of seeds induces an increase in *#ops*, since the probability that two arbitrary sessions will resemble each other decreases. Lower similarity scores imply smaller lower bounds, therefore more actions are expected to be retrieved from the metric-tree search. However, even at 16%, the performance is still better than that of I-TopK, achieving a speedup

of 3.2X (6%) to 1.6X (16%) over I-TopK.

User Idle-Times We measured how the mean user idle-time affects the performance of T-TopK. Such idle time, in which a user examines a results set and decides what action to perform next, often takes several seconds. Figure 4d depicts the performance when varying the expected time ranges from 0 to 0.24 seconds. The latter, slightly longer than the minimal human reaction time, was sufficient to compute all missing vectors offline. As expected, T-TopK improves when the idle-time increases. When all missing vectors are computed offline, see that T-TopK obtains almost a 3X speedup over I-TopK.

8 Conclusion

In this work we present an efficient computational framework for the top-k similarity search of data analysis sessions, a central component in many IDA recommender systems. By exploiting the incremental nature of the IDA sessions, our threshold-based algorithm obtains a speedup of $200X$ compared to a naive sequential search. Interesting future research opportunities are similarity bounds and algorithms to other popular measures such as edit distance and DTW, as well as investigating orthogonal directions such as approximation solutions and parallelism.

References

[1] REACT: Ida benchmark dataset. `https://github.com/TAU-DB/REACT-IDA-Recommendation-benchmark`.

[2] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi. A collaborative filtering approach for recommending olap sessions. *DSS*, 69, 2015.

[3] J. Aligon, M. Golfarelli, P. Marcel, S. Rizzi, and E. Turricchia. Similarity measures for olap sessions. *KAIS*, 39, 2014.

[4] M.-A. Aufaure, N. Kuchmann-Beauger, P. Marcel, S. Rizzi, and Y. Vanrompay. Predicting your next olap query based on recent analytical sessions. In *DaWaK*. 2013.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[6] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: interactive analytics through pen and touch. *VLDB*, 2015.

[7] M. M. Deza and E. Deza. Encyclopedia of distances. In *Encyclopedia of Distances*, pages 1–583. Springer, 2009.

[8] M. Eirinaki, S. Abraham, N. Polyzotis, and N. Shaikh. Querie: Collaborative database exploration. *TKDE*, 2014.

[9] A. Giacometti, P. Marcel, and E. Negre. *Recommending multidimensional queries*. Springer Berlin Heidelberg, 2009.

[10] J. A. Hoxmeier and C. DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000.

[11] E. Keogh and C. A. Ratanamahatana. Exact indexing of dynamic time warping. *Knowledge and information systems*, 7(3):358–386, 2005.

[12] S.-R. Kim and K. Park. A dynamic distance table. In *Annual Symposium on Combinatorial Pattern Matching*, pages 60–68. Springer, 2000.

[13] G. M. Landau, E. W. Myers, and J. P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998.

[14] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):R25, 2009.

[15] T. Milo and A. Somech. Next-step suggestions for modern interactive data analysis platforms. In *KDD*, 2018.

[16] D. Novak and P. Zezula. M-chord: a scalable distributed similarity search structure. In *ICSIS*, 2006.

[17] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference-based alignment in large sequence databases. *VLDB*, 2009.

[18] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *CIKM*, 1999.

[19] V. Roth, J. Laub, M. Kawanabe, and J. M. Buhmann. Optimal cluster preserving embedding of nonmetric proximity data. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (12):1540–1551, 2003.

[20] H. Sakoe and S. Chiba. Dynamic programming algorithm optimization for spoken word recognition. *IEEE transactions on acoustics, speech, and signal processing*, 26(1):43–49, 1978.

[21] V. Sepulveda and B. Bustos. Cp-index: using clustering and pivots for indexing non-metric spaces. In *SISAP*, 2010.

[22] T. Skopal and J. Lokoč. Nm-tree: Flexible approximate similarity search in metric and non-metric spaces. In *DEXA*, 2008.

[23] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–197, 1981.

[24] N. Sundaram, A. Turmukhametova, N. Satish, T. Mostak, P. Indyk, S. Madden, and P. Dubey. Streaming similarity search over one billion tweets using parallel locality-sensitive hashing. *VLDB*, 2013.

[25] Y. Tang, Y. Cai, N. Mamoulis, R. Cheng, et al. Earth mover's distance based similarity search at scale. *VLDB*, 2013.

[26] M. Vartak, S. Rahman, S. Madden, A. Parameswaran, and N. Polyzotis. Seedb: efficient data-driven visualization recommendations to support visual analytics. *VLDB*, 2015.

[27] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. Rcsi: Scalable similarity search in thousand (s) of genomes. *VLDB*, 2013.

[28] X. Yang, C. M. Procopiuc, and D. Srivastava. Recommending join queries via query log analysis. In *ICDE*, 2009.

[29] Y. Yuan, W. Chen, G. Han, and G. Jia. Olap4r: A top-k recommendation system for olap sessions. *KSII Transactions on Internet and Information Systems (TIIS)*, 11(6):2963–2978, 2017.

A Second Upper Bound B_2

We next describe the second upper bound for $\widehat{sup}^t(s)$ in more details, then provide an intuitive example. In Section 6.1 we stated that a second upper bound $\widehat{sup}^t(s)$ can be obtained without computing the actual prefixes similarity to u_t. We achieve this by dividing u_t into three disjoint parts, bounding separately the similarity to each part:

1. The first part consists of the prefix u_τ, from the beginning of the u_t and up to the last point in time τ, where similarity vector $\vec{Sim}(u_\tau, s)$ was computed and stored in *vecRepo*. The contribution of this part to the similarity score can be bounded by:

   $$max_{1\leq j\leq |s|}\{\vec{Sim}(u_\tau, s)[j]\}\beta^{2(t-\tau)}$$

   Namely, the maximal similarity score of the prefix u_τ of u ($\tau < t$), and a prefix of s, multiplied by a decay factor, due to the $t-\tau$ queries that were later added to the current session u.

2. The second (possibly empty) part consists of the following queries sequence, up to, but not including the last query q_t. The contribution of this part to the similarity score can be bounded by:

   $$\beta^2 \sum_{i=0}^{t-\tau-2} \beta^{2i} = \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1}$$

   Namely, the maximal alignment score for a session of size $t-\tau-1$, multiplied by the decay due to adding one query (q_t) to the ongoing session.

3. Finally, the third part consist of the last query q_t. The contribution of this part to the similarity score can be bounded by:

   $$max_{q\in s}\{\sigma(q_t, q)\}$$

   This is simply the maximal query similarity of $u_t[t]$ and any query in the session s (which has already been computed for bound B_1).

Finally, $\widehat{sup}^t(s)$ can be bounded according to Proposition 6.6:

$$\widehat{sup}^t(s) \leq max_{1\leq j\leq |s|}\{\vec{sim}_{u_\tau,s}[j]\}\beta^{2(t-\tau)} + \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1} + \\ + max_{q\in s}\{\sigma(u_t[t], q)\}$$

Example A.1. *Continuing with the running example, we now show the upper bound $B_2(\psi)$ at time $t = 5$. Assume that we last computed the similarity vector for ψ at time $t = 4$, hence $\tau = 4$. The score $B_2(\psi)$, as described above, consists of the following three parts: (i) the maximal similarity score in $\vec{Sim}(u_4, \psi)$, (ii) The maximal similarity with an empty session, since $t - \tau - 1 = 0$, and (iii) the maximal query similarity of q_5 and a query in ψ. Consequently: $B_2(\psi) = 2.09 + 0 + 1 = 3.09$*

B Additional Experiments

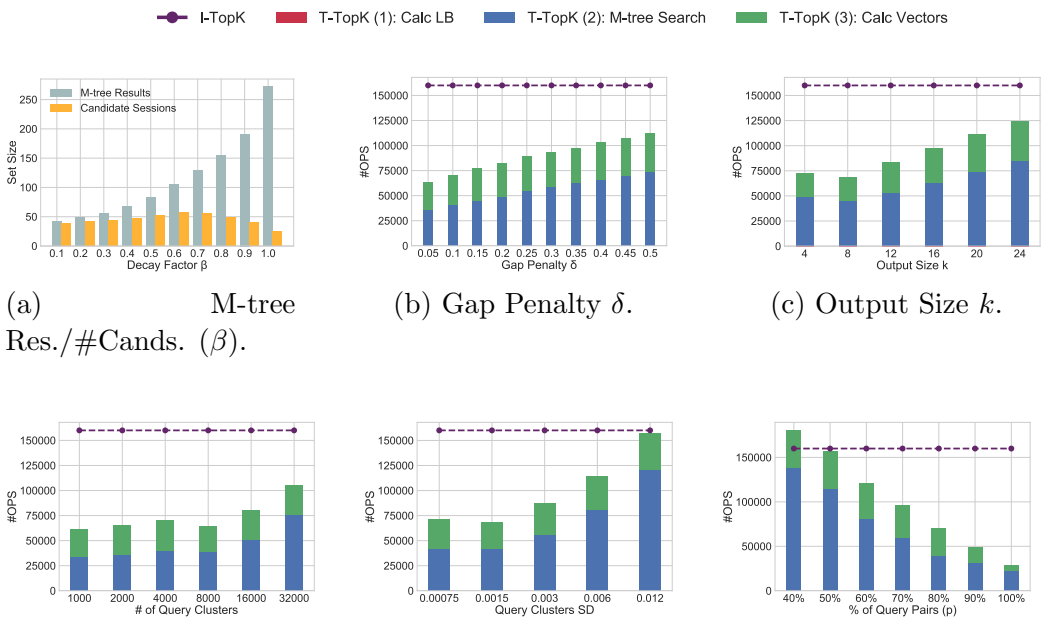
We provide additional information regarding our experimental evaluation. First, we detail the extrinsic method for determining the problem parameters. Second, we describe the construction routine for the synthetic repositories, and last, we examine the effect of the underlying characteristics of such repositories on the performance of the T-TopK algorithm.

B.1 Problem Parameters Selection

We next explain our choice of the default values for the similarity parameters, namely the decay factor β and the gap penalty δ, and how we set k, the size of the output set of T-TopK.

The values for these parameters were selected by performing extrinsic evaluation: We obtained the code of the IDA recommender system (also available in [1]) and embedded our T-TopK algorithm as its top-k search component, and performed an overall hyper-parameters selection routine, as described in [15]. Briefly, this is done by executing a grid search for the systems' parameters (+T-TopK), and selecting the configuration that allows the recommender system to achieve the highest qualitative performance. As common for recommender systems, the qualitative evaluation is performed in an offline manner: given a repository of sessions (we use the REACT-IDA repository described in Section 7.1), we simulate the sessions one by one. At each point in a session, we employ the recommender system (along with the T-TopK module) and examine whether the recommendations indeed correspond to the actual next-action performed by the user at this point. Finally, we selecting the best set of parameters (as stated in Table 1) that obtained the highest predictive accuracy and coverage[6] rates.

[6]Coverage is the percentage of situations in which the recommender system was able to provide recommendations



(a) M-tree Res./#Cands. (β).

(b) Gap Penalty δ.

(c) Output Size k.

(d) S: # of Query Clusters

(e) S: Query Distance SD.

(f) S: % Query Pairs (p)

Figure 5: Effect of Problem and Repository Parameters

B.2 Synthetic Repositories Construction

We generate a synthetic repository by first building a query (metric-) space, then constructing repository sessions by drawing queries from this space.

Generating the query space. Recall that our algorithms model queries as abstract objects in a given metric space. We represent here queries as points in an N-dimensional Euclidean space, with the Euclidean distance as the distance metric. To obtain values in $[0, 1]$ range, each element is restricted to the range $[0, \frac{1}{\sqrt{N}}]$. In common practical scenarios where queries are clustered in the metric space (e.g. group-by queries are often more similar to each other than to filtering queries). To simulate this we first draw a (controllable) number of query points uniformly at random, to serve as cluster centers, then generate additional queries around each center using a (multi-variant) normal distribution, with the cluster-center as mean. In the experiments we vary the number N of query-space dimensions, the number of query clusters, and their radius (standard deviation).

Generating repository sessions. The sessions repository is constructed in an analogous manner. We first generate a set of pivot sessions, then use them to generate sessions with varying degree of similarity to the seed. For each session (seed or other) we draw its length from a given normal distribution. To construct the pivot sessions we select, uniformly at random,

a sequence of queries of the given length. In order to control the similarity of a session to its corresponding seed, we set a percentage p of the session's queries that are drawn from the same cluster as the queries of the seed session. Specifically, for each generated session s of length l we randomly chose a number of queries $Z \backsim \mathcal{N}(p \cdot l, p^2 \cdot \sigma^2)$ where σ^2 is the same variance used for generating l. We then randomly chose Z query place-holders in s, and correspondingly Z queries in the seed session. Each place-holder in s is then assigned a query drawn (uniformly at random) from the same query-cluster as its corresponding (order-wise) query in the seed session. The remaining $(l - Z)$ queries in s were drawn uniformly at random from the full query space. Finally, in each session, for each two consecutive queries, a system idle-time was randomly generated according to a given normal distribution. In the experiments below we vary the number of sessions, their mean length, the number of session seeds, the parameter p, and the mean idle-time. After the construction was completed, we inserted all sessions' queries into the metric tree.

B.3 Effect of Problem Parameters

We next explain in details the effect of each of the problem parameters. Recall that the performance of T-TopK is divided to three segments: (1) forming the initial similarity lower bound, (2) the metric-tree search, and (3) computing similarity vectors for candidate sessions.

Decay Factor. Recall that the decay factor β reflects the relative importance of older actions in the similarity computation (lower values imply smaller importance). Since β effects the upper bound value, one expects lower values to yield more restrictive candidates selection and thereby better performance. We report the performance when using the REACT-IDA repository. Similar trends were found in the synthetic repository, thus omitted from presentation. Figure 4a in Section 7.3 displays the number of similarity computations performed by the algorithms (*#ops*) when varying the values of β from 0.1 to 1. The results for I-TopK, as mentioned above, are invariant to changes in β. We can see that in all cases the threshold-based pruning of T-TopK yields better performance, compared to I-TopK and, as expected, the advantage grows for lower values of β.

To gain a deeper insight on the performance w.r.t. the computation segments, we examine in Figure 5a, for varying β values, the number of sessions retrieved in the metric-tree search results (i.e. sessions satisfying bound B_1) and the number of sessions among them that also meet bound B_2 (hence satisfy the upper bound). We can see that the first set grows with β (and thus the cost of Segment 2 increases), but many of the retrieved

sessions are pruned via bound B_2 (thus the cost of Segment 3 does not increase). Finally, considering the running time of the algorithms, I-TopK runs in an average of 1.03ms when using the REACT-IDA repository (resp., 230ms for the synthetic), regardless of the value of β. For T-TopK, the average execution time ranges from 0.32ms (92ms) for $\beta = 0.1$, to 0.44ms (142ms) for $\beta = 1.0$ for the REACT-IDA repository (resp., the synthetic repository).

Gap Penalty. The gap penalty δ determines the score deduction due to alignment gaps. Higher (resp. lower) values thus naturally reduce (increase) the likelihood of a gap to occur but, other than that, varying the δ value had only a marginal effect on the T-TopK performance when using the REACT-IDA repository (Figure omitted). We thus focus on the synthetic repository. Figure 5b displays the performance measured in *#ops* for T-TopK and I-TopK, for a varying gap penalty value. Results are stable for $\delta > 0.5$. First, note that due to the larger size of the synthetic repository, Segment 1, i.e. computing the lower bound, is negligible, and thus hardly visible in the graphs (This holds for the rest of the evaluations on synthetic repositories). We can see some increase in the *#ops* when δ grows, particularly due to the metric-tree search (Segment 2). Intuitively, this is because when the gap penalty increases, the similarity scores generally decrease, and consequently so does the lower bound threshold inf^t. At $\delta = 0.5$ the penalty for gaps is already so high that matching is almost always preferred over gapping, consequently further increase has no additional effect.

Output Size. Recall that the output size k affects T-TopK since the similarity lower bound inf^t is computed using the top-k set of the previous iteration (see Section 6.1). Thus its computation cost increases with k, as can be seen in Figure 4b which displays the effect of k on performance (*#ops*) when using the REACT-IDA repository, as well in Figure 5c for the synthetic repository. We can see that for the larger k values, *#ops* increases due to the metric-tree range search. Intuitively, this happens because for such large k value the top-k set contains sessions with lower similarity scores, which in turn decreases the lower bound threshold.

B.4 Effect of Repository Parameters

We begin by exploring the rest of the query-space parameters, i.e., the clusters' radii and the query dimension, then examine the session space parameters, namely the number of session seeds and the ratio of similar query pairs.

Query space parameters. Recall from Table 1 that the following parameters affect the structure of the query space: *#Query Clusters*, *#Dimensions*, and the *Clusters standard deviation*. The larger their values are, the lower

the likelihood of two arbitrary queries to be similar, which generally reduces the sessions' similarity scores.

As previously explained, lower similarity scores imply smaller lower bounds, therefore more session candidates are considered in the computation. For instance, Figure 5d depicts the algorithms' performance when varying the number of query clusters from $3K$ to $32K$ (resp., 0.2% to 20% of the overall number of queries) on a logarithmic scale. In general (with minor fluctuations), the more clusters, the smaller the probability of queries (and thus of sessions) to resemble each other. Lower similarity scores imply smaller lower bounds, therefore more queries are expected to be retrieved from the metric-tree search. Indeed, *#ops* performed by T-TopK increases with the number of clusters, following an increase in Segment 2, the metric-tree search. Correspondingly, the running time of T-TopK varies from 0.1 ($3K$ clusters) to 0.14 seconds (32K clusters), comparing to 0.23 seconds obtained by I-TopK. The same principle also applies when increasing the standard deviation (std) of the distance between a query and its corresponding cluster center, see Figure 5e.

Session space parameters. Besides the query space structure, two more parameters affect the likelihood of two sessions to be similar. First is the # of session seeds, as discussed in Session 7.3. The second parameter is the *% of Query Pairs*. Recall that the parameter p determines the percentage of similar query pairs p that a session shares with its corresponding seed. Large p values imply higher similarity between sessions, and thus a higher, more effective similarity lower bound. Indeed, as depicted in Figure 5f, the performance of T-TopK improves as p increases, obtaining running times between 0.22 ($p = 0$) to 0.04 seconds ($p = 100\%$). For values under 60% I-TopK is more effective.

C Proofs

In the following, we provide formal proofs to the propositions stated in Sections 4-6.

Proof (Proposition 4.2). For every $0 \leq j \leq |s|$, by definition:

$$\vec{Sim}(u_t, s)[j] = Sim(u_t, s_j) = A_{u_t, s_j}[t, j]$$

Following Definition 3.1 we immediately obtain that for arbitrary prefixes s_i and s'_j of session s and s':

$$A_{s_i, s'_j}[i, j] = \frac{A_{s,s'}[i, j]}{\beta^{(|s|-i)+(|s'|-j)}} \tag{1}$$

By Equation 1 we get $A_{u_t,s_j}[t,j] = \frac{A_{u_t,s}[t,j]}{\beta^{|s|-j}}$,
thus $\vec{Sim}(u_t,s)[j] = \frac{A_{u_t,s}[t,j]}{\beta^{|s|-j}}$ as required. □

Proof (Proposition 4.4). We show that for the current session u_t and prefix s_j of session s the following hold:

$$\vec{Sim}(u_t,s)[j] = A_{u_t,s_j}[t,j]$$

We consider the case of $j > 1$ (the case of $j = 1$ is trivial). Expanding $\vec{Sim}(u_t,s)[j]$ and $A_{u_t,s_j}[t,j]$, we complete the proof if showing that:

$$A_{u_t,s_j}[t-1,j-1] + \sigma(u_t[t],s[j]) = \vec{Sim}(u_{t-1},s)[j-1]\beta^2 + \sigma(u_t[t],s[j])$$

$$\vec{Sim}(u_t,s)[j-1]\beta - \delta = A_{u_t,s_j}[t,j-1] - \delta$$

$$\vec{Sim}(u_{t-1},s)[j]\beta - \delta = A_{u_t,s_j}[t-1,j] - \delta$$

For the first case, by employing Equation 1 from the proof above we get:

$$A_{u_t,s_j}[t-1,j-1] = A_{u_{t-1},s_{j-1}}[t-1,j-1]\beta^2 = \vec{Sim}(u_{t-1},s)[j-1]\beta^2$$

By the same technique we can show this for the latter two equations. □

Proof (Correctness of T-TopK Algorithm). Let *top* denote the output of the algorithm. We need to show that $top_k(u_t,S) = top$. We will consider $t > 1$ (the case of t=1 is trivial).

First, we show that for an arbitrary prefix s_j of a repository session s, $s_j \in top_k(u_t,S) \rightarrow s_j \in top$. If $s \in top_k(u_t,S)$ then $Sim(u_t,s) \geq inf^t$ (as inf^t is the similarity lower bound). Hence, for the upper bound $sup^t(s)$ of session s we can easily see that $sup^t(s) \geq inf^t$, thus $s \in C$, i.e. s is retrieved and processed as a candidate session.

The proof of the case that $s_j \in top_k(u_t,S) \leftarrow s_j \in top$ follows similar lines.

□

Proof Sketch(Proposition 6.1). The proof of the left hand side inequality follows immediately from the observation that for any $S' \subseteq S$, $min\text{–}score(top_k(u_{t-1},S))\beta - \delta \leq min-score(top_k(u_t,S')) \leq min-score(top_k(u_t,S))$. In particular, the observation holds for $S' = S_{top}^{t-1}$. For the right hand side inequality, as inf^t is computed over S_{top}^{t-1}, we can show that the lowest possible inf^t is obtained when the last query in u_t is not aligned for all sessions in S_{top}^{t-1} (hence inducing a gap penalty). Thus inf^t is equal to $min-score(top_k(u_{t-1},S))\beta - \delta$ □

Proof (Proposition 6.4). If q_t is not matched with a query in s_j then: (1) $Sim(u_t, s_j) = Sim(u_{t-1}, s_j)\beta - \delta$, following Proposition 4.4. Since $s_j \notin top_k(u_{t-1}, S)$, we know that: (2) $Sim(u_{t-1}, s_j) \leq min\text{–}score(top_k(u_{t-1}, S))$. From (1) and (2) We get $Sim(u_t, s_j) \leq min\text{–}score(top_k(u_{t-1}, S))\beta - \delta \leq inf^t$ The latter transition stems from the observation that $min\text{–}score(top_k(u_{t-1}, S))\beta - \delta$ is a lower bound on inf^t. □

Proof Sketch (Proposition 6.5). Let s_m be the prefix of s with the maximal similarity to u_t. Recall that following Proposition 6.4 we assume that q_t aligns with a query in s, and using Proposition 4.4, we can prove that q_t is matched with its last query $s_m[m]$. Consequently,

$$Sim(u_t, s_m) = Sim(u_{t-1}, s_{m-1})\beta^2 + \sigma(q_t, s_m[m])$$

We know that $Sim(u_{t-1}, s_{m-1}) \leq min\text{–}score(top_k(u_{t-1}, S))$, and $\sigma(q_t, s_m[m]) \leq max_{q \in s}\{\sigma(q_t, q)\}$, thus the proposition holds. □

Proof (Proposition 6.6). We prove by induction. As shorthand, we denote $s_\tau := max_{1 \leq j \leq |s|}\{\vec{sim}_{u_\tau, s}[j]\}$. The base case is when $t = \tau + 1$. We can easily show that:

$$\widehat{sup}^{\tau+1}(s) \leq s_\tau\beta^2 + max_{q \in s}\{\sigma(u_t[t], q)\}$$

We know that:

$$\widehat{sup}^{t+1}(s) \leq \widehat{sup}^{t}(s)\beta^2 + max_{q \in s}\{\sigma(u_{t+1}[t+1], q)\} \tag{2}$$

Assuming the inequality holds for t, we can bound $\widehat{sup}^{t}(s)\beta^2$:

$$\widehat{sup}^{t}(s)\beta^2 \leq \beta^2(s_\tau\beta^{2(t-\tau)} + \frac{\beta^{2(t-\tau)} - \beta^2}{\beta^2 - 1} + max_{q \in s}\{\sigma(u_t[t], q)\})$$

Since $max_{q \in s}\{\sigma(u_t[t], q)\} \leq 1$ we have:

$$\widehat{sup}^{t}(s)\beta^2 \leq s_\tau\beta^{2(t+1-\tau)} + \frac{\beta^{2(t+1-\tau)} - \beta^4 + \beta^2(\beta^2 - 1)}{\beta^2 - 1}$$

We can use the latter expression in (2) to obtain:

$$\widehat{sup}^{t+1}(s) \leq s_\tau\beta^{2(t+1-\tau)} + \frac{\beta^{2(t+1-\tau)} - \beta^2}{\beta^2 - 1} + max_{q \in s}\{\sigma(u_{t+1}[t+1], q)\}$$

Hence the inequality holds for $t + 1$ □