Technical Report: Incremental Top-k Similarity Search for Interactive Data-Analysis Sessions

Interactive Data Analysis (IDA) is a process in which users explore data in an incremental manner, issuing a sequence of queries, 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-query to issue at each point in the session. Such recommender systems often record user sessions in a repository and utilize them to generate next-query 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 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 algorithms that exploit this incremental nature of IDA sessions to speed up the search. We first provide a simple generic model for describing IDA user sessions and measuring their similarity via an alignment matrix. 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 optimization techniques. Our experiments, on both real-life and synthetic data, demonstrate the efficiency and effectiveness of our solution.

1. INTRODUCTION

Interactive Data analysis (IDA) is fundamentally an iterative process in which a user issues a query, receives a results set, and decides if and which query to issue next. A sequence of queries issued by a user is called a *session*. IDA is a challenging process, especially to inexperienced users. Therefore, extensive research has been done, developing recommender systems that assist users in choosing, at each point in the session, an appropriate next-query to issue [20, 13, 2, 17, 3]. Such recommender systems often take a *collaborative-filtering* approach and use a repository of (prior) analysis sessions of the same or other users to generate recommendations. Following the assumption that if two session prefixes are similar, their continuation is likely to also be similar, these systems retrieve the top-k session prefixes most similar to the current user session, examine their continuation and use the gathered information to form a possible next-query recommendation [16, 17, 39, 3].

Proceedings of the VLDB Endowment, Vol. 10, No. 7
Copyright 2017 VLDB Endowment 2150-8097/17/03.

The retrieval of the top-k similar session prefixes is a central component in IDA recommender systems. Since the sessions repository may be large, efficiency is critical here, 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 *at each step*. Therefore, the question that we ask is whether the incremental nature of IDA sessions can be exploited to speed up the search. Given a sessions repository and an ongoing user session, we call the problem of iteratively computing the top-k most similar session prefixes, the *incremental top-k similarity search problem*. The goal of this paper is to devise efficient algorithms to solve this problem.

In this work we employ a comprehensive similarity measure that was defined in [4] based on the well known Smith-Waterman Algorithm [33]. In the comparative study of [4] this measure was found to be most suitable, quality-wise, for the context of analysis sessions. Intuitively, given some similarity metrics for individual queries, the algorithm aligns two sessions s, s' by matching similar query pairs. Gaps are allowed (yet penalized) and more weight, score-wise, is given to the more recent queries in the sequence, since they are expected to have higher relevance on the current user's intent. More formally, the best alignment of two sessions s and s' (and its corresponding score) is defined, inductively, via an alignment matrix, whose (i, j) cell determines the best alignment of the i-length and j-length prefixes of s and s', resp., based on the alignment of their shorter prefixes (see definition in the sequel).

While a large body of previous research suggests optimizations for general sequence alignment, mostly in the context of bioinformatics [24, 18, 38], most work focus on a "one time" similarity search (rather than a repetitive search w.r.t. an incrementally growing sequence). Furthermore, these solutions often assume a fixed, small alphabet (e.g. nucleotides) while in our context the "alphabet" (namely the query space) is unbounded, and there exist a distance metric between letters (queries). To the best of our knowledge, our work is the first to investigate the incremental top-k similarity search problem that is typical to IDA systems. In particular, we harness the incremental nature of IDA sessions to speed up the top-k similarity search.

Let S be a repository of sessions and let $u = q_1, q_2, \dots$ be the ongoing user session. At each time $t = 1, 2, \dots$ we have the prefix $u_t = q_1 \dots q_t$ at hand, and we need to retrieve from S the top-k session prefixes most similar to u_t. A naive brute-force solution is to iterate over all sessions s in the repository, compute the similarity score of u_t w.r.t. each prefix of s , then select the k prefixes with the highest score. The key insight underlying our optimized algorithm is that the inductive nature of the alignment process tally with the iterative nature of the top-k retrieval, thereby facilitating optimization along three dimensions. The first two, rather direct, optimizations are inspired by previous work on string similarity [22, 21] whereas the third requires heavier, novel machinery. We intuitively explain this next.

Optimized algorithm. Consider a user session u_t at time t. The first immediate observation is that for each repository session s, rather than examining each of its prefixes separately, it suffices to compute the alignment matrix of u_t and s *only once*, then derive from it the similarity scores of all the session prefixes. This is because, as mentioned above, the matrix already includes information about the prefixes alignment and only minimal additional processing is thus required. The second observation, more important to our context, is that it is also possible to reuse the computations performed at time $< t$ to speed up performance at time t. Specifically, we show that due to the same inductive nature of the alignment process, similarity scores for u_t, the user session at time t, can be efficiently derived from similarity scores computed in the previous iteration at time $t-1$, for the prefix u_{t-1}, refined by the new knowledge on the additional action added at time t.

Finally, and most challenging, we show that the information derived in the previous iterations can further be used to prune the search space and examine only a restricted, relatively small, portion of the sessions repository. The crux of the solution is an incremental, threshold-based algorithm that employs two types of bounds, both derived from previous-iteration data, that allow to significantly reduce the search space of the algorithm. The first is a *global lower bound* for the similarity score of a prefix 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 need to be considered. 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 criteria.

Deriving effective similarity bounds. 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 query added to the user session, to derive relaxed bounds, close enough to the exact ones.

Efficiently identifying candidate sessions. To allow for efficient computation one should be able to identify relevant sessions *without explicitly iterating over all sessions in the repository* and computing their bounds. To that end we analyze the properties of the bounds and derive conditions on the individual queries from which relevant sessions are composed. These conditions then facilitate the use of a dedicated index structure (a metric tree) for fast session retrieval. Our algorithm (to be detailed in the following sections) uses the metric-tree to quickly prune irrelevant sessions. The full bound is then computed only for the selected sessions, to further prune the candidates set.

Utilizing system idle-time. Our optimized threshold-based algorithm computes, at each time t, the similarity scores only for the selected candidate sections. As this selected set can vary over time, some of the sessions might have not been considered in previous iterations which may hinder the incremental computation. Our algorithm allows utilizing system idle-times to complete missing data and speed run-time performance.

We demonstrate the efficiency of our solution through an extensive set of experiments, on both real-life and synthetic data. We use the real-life data to demonstrate the efficiency of our algorithm in practical settings and highlight benefits of our threshold-based optimization. We use the synthetic datasets to vary the main parameters that may affect the algorithm's performance and demonstrate its resilience.

The technical contributions and the paper organization are as follows:

— In Section 2, we provide a simple generic model for describing IDA sessions, coupled with an alignment-based session similarity notion. Based on this model, we develop a formal definition for the incremental top-k retrieval problem.

— In Section 3 we show how the iterative nature of the similarity search is exploited to speed up the calculation of the alignment matrix, and then present two complementary algorithms for the incremental top-k search problem.

— In Section 4 we devise effective similarity lower and upper bounds, also stemming from the iterative nature of the top-k search, and show how to efficiently use them to prune irrelevant sessions.

— In Section 5 we demonstrate the efficiency of our solution via an extensive experimental evaluation.

We review related work is in Section 6, and conclude in Section 7. For readbility, note that we defer the proofs for all theorems and propositions throughout this work to Appendix A.

2. PRELIMINARIES

We start by providing the basic definitions for IDA sessions, then define the incremental top-k similarity search problem that we study in this paper.

Interactive Analysis Sessions. In the process of IDA, users investigate a dataset(s) via an interactive user interface (UI) which allows them to formulate and issue queries and to examine their result sets.

Formally, we assume an infinite domain of analysis queries $\mathcal{Q}$ and we model an analysis session as a sequence of queries $s = \langle q_1, q_2, \dots, q_n \rangle | q_i \in \mathcal{Q}$. We use $s[i]$ to denote q_i, the i'th query in s and s_i to denote the session's *prefix* up to q_i, i.e. $s_i = \langle q_1, q_2, \dots, q_i \rangle$. With this notation $s = s_n$ and s_0 is the empty session.

The user session is built incrementally. At time t the user issues a query q_t, then analyzes its results and decides whether to issue a next query q_{t+1} or to terminate the session. The session (prefix) u_t at time t thus consists of the queries $\langle q_1, \ldots, q_t \rangle$, where the following queries 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. $\mathcal{P}refix(S) = \{s_i \mid s \in S, 1 \leq i \leq |s|\}$.

As explained in the introduction, to assist the user in choosing an appropriate next-query, 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-query recommendation. To formally define sessions similarity, let us first consider the similarity of individual queries, then generalize to sessions.

Query Similarity. Given a *query distance metric* $\Delta : \mathcal{Q} \times \mathcal{Q} \rightarrow [0, 1]$ over the queries domain, the *query 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 query distance measures (and correspondingly similarity functions) have been proposed in the literature (e.g. [27, 6, 17, 4]) and our framework can employ any of them as long as the corresponding query distance defines a metric space.

Session Similarity. There are many ways to lift the similarity of individual items into similarity of item sequences [12] (e.g. using edit-distance, Dice coefficient, and Hausdorff distance). For query sessions, the authors of [4] formulated a desiderata for sessions similarity in the context of Online Analytical Processing (OLAP), based on a comprehensive user study: (1) It should take the queries *order of execution* into consideration, i.e. two sessions are similar if they contains a *similar* set of queries, performed in a *similar order*. (2) "Gaps" (i.e subsequences of non-matching queries) should be allowed yet penalized. (3) Long matching subsequences should be rewarded. (4) Recent queries are more relevant than old queries. Following this desiderata they refined the *Smith-Waterman similarity measure* [33], a popular measure for local sequence alignment, and proposed a similarity measure for query sequences, which we adapt to our context.

We start by describing the sessions similarity measure that we use, then explain the adaptations made w.r.t [4]. Intuitively, 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 queries at the end of the considered prefixes is considered, and the best option, score-wise, is chosen: The two queries may either be matched, in which case an award proportional to their similarity is added to the score accumulated for their preceding prefixes, or, alternatively, one of the queries is skipped over, moving to match the next query in the session, in which case a *gap penalty* $0 \leq \delta \leq 1$ is deducted from the accumulated score. To reflect the fact that the matching/skipping of older queries is less important than that of recent ones, the rewards/penalties are multiplied by a decay factor $0 \leq \beta \leq 1$ with an exponent reflecting how distant the queries are from the sessions' end.

		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 $u_4 =$ "$ABAB$"

Formally, to define sessions similarity we use an *alignment matrix* defined as follows.

Definition 2.1 (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:

$$Sim(s, s') := A_{s,s'}[n, m] \tag{1}$$

To illustrate the session similarity notion, consider the following example.

Example 2.2. *For simplicity, assume that the query space is represented by the English letters, both uppercase and lowercase, and assume the following query similarity for any two distinct uppercase letters X,Y and their corresponding lowercase versions x,y.*

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

Intuitively, identical letters are maximally similar; Two instances of the same letter, but in different case, are 0.5 similar; and different letters of the same case are 0.1 similar.

Let $u_4 =$"ABAB" be the current user session, and consider two repository sessions $\phi =$"abcAB" and $\psi =$"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 queries are matched. As we can see, the similarity score $Sim(u_4, \phi)$ is higher than $Sim(u_4, \psi)$.

As mentioned, our similarity definition follows that of [4] with some small adaptations to our context. First, we define the similarity score as the bottom-right cell of the alignment matrix, as opposed to the maximal matrix cell value in [4, 33]. This is because we focus here on measuring the similarity of two sessions - the user session executed so far and

u \ ϕ		a	ab	abc	acbA	abcAB
ABAB	0	0.35	0.91	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

the repository session prefix that is being examined - rather than generally matching arbitrary sub-sessions. Second, we apply a decay factor to both query matches and gaps, as opposed to only matches in [4, 33], since both lose significance as the session advances. The decay factor and gap penalty parameters here are constants that can be determined by the system administrator, e.g. via data analysis. The measure of [4] allows for using more complex setting parameters whose values may dynamically change at run time. As we shall see, the use of constant parameter values facilitates effective optimization via computation factorization. We leave the investigation of optimization in the presence of dynamic parameter settings to future research.

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,

Definition 2.3 (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 $top_k(u_t, S)$.*

3. INCREMENTAL TOP-K SEARCH

To solve the top-k similarity search problem we need to find, iteratively, the set $top_k(u_t, S)$, for $t = 1, \ldots, |u|$. A naive solution is to iterate over all sessions in the repository S, and compute for each prefix its alignment matrix with u_t, then select the k prefixes with the highest similarity score. Our solution improves over this naive algorithm along three dimensions: (1) We show that it is sufficient to compute one alignment matrix between two sessions, then use it to derive *all prefixes similarity scores.* (2) Exploiting the iterative nature of the problem, we show that the alignment matrices of u_t, the user session at time t, can be efficiently derived from the matrices computed at $t-1$. (3) We present a novel threshold-based algorithm for the top-k search, facilitating the use of pruning technique to effectively reduce the search space. We next describe each of these dimensions.

3.1 Computing Similarity Vectors

Given u_t - the user session at time t - and a session $s \in S$, the alignment matrix $A_{u_t,s}$ is used to compute the similarity score $Sim(u_t, s)$ of u_t and s, as shown in the previous section. We will next show that the same matrix can be used to compute a *similarity vector*, $\vec{Sim}(u_t, s)$, recording the scores of u_t and each of s's prefixes. I.e.,

$\vec{Sim}(u_t, s) = [Sim(u_t, s_0), Sim(u_t, s_1), \ldots, Sim(u_t, s_{|s|})]$

In the following we use $\vec{Sim}(u_t, s)[j]$, $j = 1 \ldots |S|$, to denote the j element in the vector. We employ the following observation:

Observation 3.1. *Given two session s, s' and an alignment matrix $A_{s,s'}$ for s and s', the similarity score of any two session prefixes s_i, s'_j is given by:*

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

More generally, following Definition 2.1, we can show that:

$$A_{s_i,s'_j}[i, j] = \frac{A_{s,s'}[i, j]}{\beta^{(|s|-i)+(|s'|-j)}} \qquad (2)$$

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

Proposition 3.2 (Similarity Vector). *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 3.3. *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 3.2, 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: $\vec{Sim}(u_4, \phi)[3] = \frac{A_{u_4,\phi}[4,3]}{\beta^2} = \frac{0.58}{0.81} = 0.71$.*

The time complexity of computing the similarity vector of u_t w.r.t. s is thus reduced to the complexity of computing the corresponding alignment matrix. If $|\widehat{s}|$ is the average session size and λ is the complexity of computing similarity for individual queries, this amounts to an average complexity of $O(|\widehat{s}|^2\lambda)$. We shall next see how this can be further reduced by exploiting the incremental nature of the computation.

3.2 Incremental similarity computation

We next explain how similarity vectors at time t can be efficiently computed from the corresponding vectors at time $t-1$. Namely how for every session s, $\vec{Sim}(u_t, s)$ can be computed from $\vec{Sim}(u_{t-1}, s)$.

Theorem 3.4. *For every repository session s, user session u and time $t > 1$,*

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

For proof, refer to Appendix [?]

Example 3.5. *Continuing with our example, assume that the user now issues a new query "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 3.4 we can derive each value in the new vectors from the previous corresponding similarity vectors at time $t = 4$.*

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

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

Using Theorem 3.4 we can derive a simple algorithm for computing the top-k most similar sessions at time t. (We will further improve this algorithm in the next section.) The Algorithm, denoted I-TopK, is depicted in Algorithm 1. It uses a repository named *vecRepo* of similarity vectors computed in previous iterations. In the algorithm we use $vecRepo[u_{t'}, s]$ to denote the stored similarity vector of $u_{t'}$, $t' < t$, w.r.t. session s.

I-TopKtakes 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 Theorem 3.4.

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:     $\vec{Sim}(u_t, s) \leftarrow ComputeVector(vecRepo, u_t, s)$
4:     for $(j = 1; j \leq |s|; j++)$ do
5:         $top.push(\vec{Sim}(u_t, s)[j], s_j)$
6: return $top$
```

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 vecotr size $(|s|+1)$ of zeros.
3: for $(j = 1; j \leq |s|; j++)$ do
4:     Compute $\vec{Sim}(u_t, s)[j]$ as in Theorem 3.4
5: $vecRepo[u_t, s] \leftarrow \vec{Sim}(u_t, s)$
6: Return $\vec{Sim}(u_t, s)$
```

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 can be deleted. Regarding the time complexity of the algorithm, observe that here the average time complexity of computing the similarity vector of u_t w.r.t to s is reduced to $O(|\hat{s}|\lambda)$,where $|\hat{s}|$ is the average session size and λ is the complexity of computing query similarity. The full top-k computation at time t iterates over the repository sessions and thus takes on average $O(|S||\hat{s}|\lambda)$. We will next show how to avoid scanning the full repository and instead focus only on the "potentially promising" candidates.

3.3 Threshold-Based Algorithm

Our optimized algorithm harnesses natural lower and upper bounds of similarity scores to facilitate pruning of unpromising sessions, while deferring some of the computation to system idle-times. In what comes next, we present the general workflow of the algorithm. Then in Section 4 we describe an efficient implementation for its different components. The algorithm consists of six key components.

1. Forming a 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. Note that an exact *tight* threshold may be computed only after the top-k set is known, as the minimum similarity score in $top_k(u_t, S)$. As an approximation we will have our inf^t smaller but 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 $sup^t(s)$. To compute an exact *tight* upper bound, one would need to compute the actual similarity scores for each of s's prefixes, and take the maximum. We thus devise an upper bound $sup^t(s)$ 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 potential to enter the top-k set. We call these the *candidate sessions*.

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 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. The missing similarity vectors thus must be completed before the $\vec{Sim}(u_t, s)$ can be computed.

5. Online vs. Offline computation. The missing similarity vectors for a session s may be computed at run-time, i.e. when they are needed to compute $\vec{Sim}(u_t, s)$. Alternatively, to speed up computation, the system can use its idle-time to complete the missing data offline. Note that in data exploration systems there is typically some idle-time between the execution of consecutive queries, in which the analyst examines the results of her query before deciding on the next action. Our algorithm allows utilizing such idle-times, to compute offline as many missing vectors as possible (depending on how much idle-time there is), so that they are already available when needed.

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 from it 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, it retrieves the prefixes set $top_k(u_t, S)$ as follows: First, if u_t contains only one query, 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 a query 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*.

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 $(j = 1; j \leq |s|; j++)$ 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
```

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

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 $O(\alpha + \lambda |\widehat{C}| \widehat{\tau} |\widehat{s}|)$. 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?" naturally depends on how small α, $|C|$ and $\widehat{\tau}$ are.

In the following section, we complete the abstract T-TopK framework with efficient implementations for the similarity lower- and upper-bounds, both derived from the previous iteration data. Later, in Section 5, we perform an empirical analysis of the performance, demonstrating that the complete T-TopK framework outperforms I-TopK in realistic scenarios.

4. LOWER AND UPPER BOUNDS

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

4.1 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 $top_k(u_t, S)$. As mentioned, an exact, tight lower bound would simply be the minimal similarity score of a prefix in this set, namely $min-score(top_k(u_t, S))$. But since computing this value requires already knowing the top-k set, we use a looser (yet still effective) threshold that can be computed efficiently.

Intuitively, we follow the assumption that underlies common recommender systems, that if two session prefixes are similar, their continuation is likely to be similar as well. Consequently, we use the sessions in the (already computed) top-k set of time $t-1$ as potential representative for the top-k set of time t, and define our threshold w.r.t them.

More formally, let $S_{top}^{t-1} \subseteq S$ 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_{top}^{t-1}, and define our threshold as:

$$inf^t = min-score(top_k(u_t, S_{top}^{t-1}))$$

Note that since the size of S_{top}^{t-1} 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 4.1. $min\text{–}score(top_k(u_{t-1}, S))\beta - \delta \leq inf^t \leq min-score(top_k(u_t, S))$

For illustration, consider the following example:

EXAMPLE 4.2. *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 3.3), 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 $S_{top}^4 = \{\phi\}$. For computing the threshold $inf^5 = min\text{–}score(top_1(u_5, S_{top}^4)$, 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 $inf^5 = 1.95$.*

4.2 Upper-Bounding the Similarity Scores

We next explain how the $sup^t(s)$ upper bounds are defined and, no less importantly, how the candidate sessions - whose upper bound $sup^t(s)$ is greater than inf^t - can be efficiently identified (in line 6 of Algorithm 3).

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. Note that for the sessions in S^{t-1}_{top} (the sessions with prefixes in $top_k(u_{t-1}, S)$), the information required to compute $\widehat{sup}^t(s)$ is already at hand, as it was obtained when computing the similarity lower bound, inf^t. Thus for each session $s \in S^{t-1}_{top}$ we can set $sup^t(s) = \widehat{sup}^t(s)$ with practically no cost. However, for the remaining sessions $s \in S \setminus S^{t-1}_{top}$, computing a tight upper bound is expensive. Therefore we use for them a looser $sup^t(s)$ value, that bounds $\widehat{sup}^t(s)$ from above.

The key observation underlying our solution is that it suffices to restrict our attention to sessions having some prefix whose similarity to u_t is above the threshold inf^t.

Definition 4.3 (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* $Sim(u_t, sj) > inf^t$*, 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 intuitively 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. Hence, the algorithm is still sound even if their proper upper bound is lower than the actual score.

To complete the presentation we explain below (1) how our proper upper bounds are defined, and (2) how candidate sessions that satisfy the bound can be efficiently identified.

4.2.1 Defining the Bound

Recall from Definition 2.1 that to compute the similarity between u_t and s we align the two sessions. In particular, for each query in $q \in u_t$ we decide, in the alignment process, whether it is beneficial to match it with a query $q' \in s$ (in which case a bonus proportional to the queries similarity is added to the score), or to skip it (in which case a gap penalty δ is deducted from the score). Let q_t be the last query in the session u_t. The following proposition shows that, when defining a proper upper bound, one only needs to consider sessions where q_t is matched to one of their contained queries.

Proposition 4.4. *For a session* $s \in S \setminus S^{t-1}_{top}$*, if when computing* $Sim(u_t, s)$ *the query* q_t *is not matched with any of* s*'s queries, then for every prefix* s_j *of* s*,* $Sim(u_t, s_j) \leq inf^t$

Consequently, in the analysis below we ignore sessions where q_t is not matched with any of s's queries, and for brevity, unless stated otherwise, whenever we refer to a session s we mean one in $S \setminus S^{t-1}_{top}$ 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 queries to the new query q_t.

Proposition 4.5. *For every session* s*,* $\widehat{sup}^t(s) \leq min{-}score(top_k(u_{t-1}, S))\beta^2 + max_{q \in s}\{\sigma(q_t, q)\}$

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 $min{-}score(top_k(u_{t-1}, S)$. Consequently, even if the newly added query q_t is matched to the most similar query 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 query) 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 queries in s. As the queries reside in a metric space, we show in Section 4.2.2 how this is done efficiently.

Example 4.6. *For our running example, at* $t = 5$ *the upper bound* $B_1(\psi)$ *is the sum of two values: (i) the minimal similarity score in the top-k at* $t = 4$*, (i.e.* $\vec{Sim}(u_4, \phi)[5] = 2.28$*), and (ii) the maximal query similarity between* q_5 *("c") and a query in* ψ*. I.e.,* $B_1(\psi) = min{-}score(top_1(u_4, \{\phi, \psi\}))\beta^2 + max_{q \in \psi}\{\sigma(\text{“c”}, q)\} = 2.28 * 0.81 + 1 = 2.84$

Second bound (B_2)***.*** An alternative way to bound $\widehat{sup}^t(s)$, without computing the actual prefixes similarity to u_t, is 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).

Putting everything together we get the following proposition, and set our second bound B_2 as the right hand side expression of the inequality.

Proposition 4.7.

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

Example 4.8. *Continuing with the 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:* $B_2(\psi) = 2.09 + 0 + 1 = 3.09$

Note that the computation of B_2 requires no query 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. Namely, $sup^t(s) = min(B_1(s), B_2(s))$

Example 4.9. *Consider again session ψ at $t = 5$. Its upper bound is given by $sup^5(\psi) = min(\{B_1(\psi), B_2(\psi)\}) = min(2.84, 3.09) = 2.84$. Recall from Example 4.2 that the lower bound $inf^5 = 1.95$. ψ thus meets the bound condition and is considered a candidate session. Indeed, later on, ψ_5 is chosen as the top-1 similar prefix to u_5.*

4.2.2 Efficient Retrieval of Candidate Sessions

As mentioned above, we use the first bound B_1 to quickly identify potentially 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 4.5.

Observation 4.10. *For $sup^t(s)$ to be not smaller than inf^t, s must contain some query 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 queries, we employ an index structure that uses the fact that the query similarity measure defines a *metric space* (See Section 2). Specifically, in our implementation we use a *metric-tree* [10] - an index that harnesses the triangle inequality property of a metric space to facilitate a fast similarity search. For completeness of presentation we briefly overview the main characteristics of metric-trees (for full details see, e.g., [10]), then explain how they are employed in our setting.

Metric-trees. Briefly, a metric-tree is a tree structure, constructed such that all objects are stored in its leaves. Non-leaf nodes comprise of (1) a representative object chosen among its descendant leaf nodes, (2) a "covering" radius, denoting the maximal distance between the representative object and all of its descendants, and (3) for each of its children nodes, the distance between the representative object of the node and that of its child, as well as the child's covering radius. The metric-tree supports fast *range search*, i.e. given an object o and a distance threshold θ, it allows to retrieve all indexed objects o' which are distant at most θ from o, i.e. $distance(o, o') \leq \theta$. The search process works as follows: Starting from the root node, at any non-leaf node we compute the distance between the query object o and the representative object of that node. By employing the triangle inequality property, traversal paths can be pruned from the search: let v_1 and v_2 be non-leaf nodes having representative objects o_1 and o_2 resp., where v_2 is a child of v_1. The search "skips" $v2$ if $d(o, o_1) + d(o, o_2) > \theta + r(v_2)$, where $r(v_2)$ denotes the covering radius of v_2.

Sessions selection via the queries metric-tree. In our context, we maintain a metric-tree recording the sessions queries, with pointers from each query to the session it appears in[1]. We use the query distance (the complement of their similarity) as the distance function for the tree, i.e. $\Delta(q, q') = 1 - \sigma(q, q')$. Note that from Observation 4.10 and our definition of the distance function, it follows that all sessions that satisfy the bound B_1 must contain some query 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 queries 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 4.11. *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 4.2). Recall that our goal here is to retrieve sessions having a query q s.t. $\Delta(q_5, q) \leq 1 - inf^5 + min{-}score(top_1(u_4, \{\phi, \psi\}))\beta^2$, namely queries 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 B_2 is computed, as in Example 4.9 to obtain the final set of candidate sessions.*

We conclude this section with two remarks, one about the cost of the algorithm and the second about parallelism. First, recall from Section 3.3 that the efficiency of the T-TopK algorithm depends, among others, on α, the cost of identifying the candidate sessions. From the discussion above this is determined by the cost of the metric tree search, which in turn depends on the number of query similarity operations performed, times the cost λ of such individual operation (which is a black-box in our framework). In the following section we conduct a thorough empirical study showing that in realistic settings α is small enough, hence T-TopK mostly outperforms I-TopK.

[1]Syntactically-identical queries issued in different sessions are stored separately

Parameter	**Min**	**Max**	**Default**
Algorithm Parameters			
Decay Factor	0.1	1	0.9
Gap Penalty	0.05	1	0.1
Output Size	4	24(64)	16
Controlled Dataset Parameters			
#Sessions	1K	100K	10K
Session len.	$\mathcal{N}(4,3^2)$	$\mathcal{N}(32,12^2)$	$\mathcal{N}(16,3^2)$
#Query Clusters	1K	32K	6K
Cluster Rad. (std)	0.00075	0.012	0.003
Query Dim.	5	500	25
#Session Seeds	6%	16%	10%
%Query Pairs (p)	0%	100%	80%
Offline Quota	0	$\mathcal{N}(168\text{K},34\text{K})$	$\mathcal{N}(120\text{K},24\text{K})$

Table 1: Similarity Algorithm and Dataset Parameters

Second, we note that both the I-TopK and the T-TopK algorithms may be nicely parallelized in a multi-core environment: The repository may be split across the cores with each core responsible for computing the alignments to u_t for its share. Parallelizing the full T-TopK algorithm requires some extra effort due to the use of the metric-tree. To do so, one may use metric-tree extensions or alternative frameworks (e.g. [42, 29, 37]), designed for parallelized similarity search in metric spaces.

5. EXPERIMENTAL STUDY

We have 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 Java implementation (`https://github.com/erdavila/M-Tree`), All experiments were conducted over *Intel Core i7-4790, 3.6GHz* machine (4 dual cores), equipped with 8GB RAM and running *Windows 7*.

To the best of our knowledge, our work is the first to investigate the incremental top-k similarity search problem, that is typical to IDA systems. State-of-the-art sequence alignment solutions (stemming from the bioinformatics domain) consider a limited small-size alphabet. This is incompatible with our setting where the query-space is unbounded, with a given distance metric between letters (queries). General solutions for search in non-metric space[2] require to index all prefixes of all sessions, whose number may be prohibitively large (typically an order of magnitude larger than the number of sessions). We thus examine the performance of T-TopK and I-TopK compared to (1) the naive solution mentioned in the Introduction, as well as (2) a semi-naive variant that computes a single similarity matrix per repository session (as described in Section 3.1). As they both compute the alignment matrices from scratch at each step, their performance is significantly inferior. We thus omit their results and focus on the comparison between T-TopK and I-TopK.

We conducted two sets of experiments. The first uses real-life data to evaluate performance in real-life settings, whereas the second uses a synthetic sessions repository to vary the data characteristics and scale.

5.1 Real-life data

We are not aware of commercial analysis platforms that publish customers' query logs. We have thus generated a real-life sessions repository by recruiting 56 analysts and asked them to use the REACT-UI [28], an interactive analysis platform developed by a subset of the authors, to analyze each four different datasets from the domain of cyber-security [34]. We used the recorded analysis sessions as our session repository. The average session length was 8.5 queries and the repository contains a total of 1100 distinct queries. The median system idle-time, i.e. the time between two consecutive queries, was 40 seconds, which sufficed for computing all missing similarity vectors.

Methodology and Evaluation. We examined in this experiment the effect of the problem parameters, namely the gap penalty δ, the decay factor β and the output set size k, on the algorithms behavior. In each experiment, we chose values for δ, β and k, then drew, uniformly at random, a repository session as the current ongoing session u and withdrew it from the repository. Next, we solved the incremental top-k search problem for u, with both T-TopK and I-TopK, using REACT's query similarity function as the query similarity measure, and the time measured between consecutive queries in u as the system idle time. For each setting, we repeated the experiments 100 times.

Recall from our analysis of the I-TopK and the T-TopK algorithms that in both the performance essentially depends on the number of query similarity computations performed and the average cost λ of such an individual computation (which is the same in both). λ may vary depending on the specific similarity function being used. Thus to properly compare the algorithm performance, rather than measuring the running time, we report the average (per-step) number of similarity computations performed by each algorithm (denoted *#ops*). Then, we state the approximated running time by assigning λ the time required for the query similarity function implemented in the REACT system [3].

Results. We present below selected results when varying a single parameter (δ, β or k) and setting the rest to a default value corresponding to a typical usage scenario, as depicted in Table 1. Note that the *#ops* performed by I-TopK is not dependent on the values of δ, β and k. Its performance is thus represented in the graphs by a horizontal line.

Decay Factor. Recall that the decay factor β reflects the relative importance of older queries in the similarity computation (lower values imply smaller importance). Since β effects the upper bound value, one expects lower values to yield more selective candidates selection and thereby better performance. This is indeed demonstrated in Figure 3a that displays the number of similarity computations that each of the algorithms (*#ops*) for varying the values of β (from 0.1 to 1). The results for I-TopK (the blue horizontal line) are invariant to changes in β.

For T-TopK we show the *#ops* for each value of β segmented into three parts: (1) the *#ops* required to form the initial similarity lower bound, (2) the *#ops* performed in the course of the metric-tree search, and (3) the *#ops* required for computing similarity vectors for candidate sessions. Note that in the latter we do not count the offline work as it does not affect the response time. We can see that in all cases the threshold-based pruning of T-TopK yields better perfor-

[2]Note that while *query* similarity forms a metric space, the alignment-based *sessions* similarity does not.

[3]The similarity function takes into consideration both the query syntax and (a signature of) the results. For details see [28]

mance, compared to I-TopK and, as expected, the advantage grows for lower values of β.

We can see that Segment 1 remains unchanged here, because the lower bound computation cost is independent of the value of β. We can also see that Segment 2 grows when β increases, whereas Segment 3 stays almost unchanged. To gain a deeper insight on this behavior, we examine in Figure 3b, 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 do not increase).

Finally, considering the running time of the algorithms, I-TopK runs here on average in 47ms regardless of the value of β. For T-TopK the average execution time ranges from 14ms for $\beta = 0.1$ to 27ms for $\beta = 1.0$. (We omit the graph for space constraints).

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 very marginal effect on the T-TopK performance here. Figure omitted. Here too T-TopK performs significantly better than I-TopK with an average of 626 query similarity operations (and running time of 25ms), compared to 1094 operations (and 47ms) for I-TopK.

Output Size. 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 4.1). Thus its computation cost increases with k, as can be seen in Figure 3c. The figure depicts the *#ops* for k values ranging from 1 to 25 (common value for data analysis recommender systems is below 20 [16, 4, 28]). Each bar is split into the same three segments as above. As expected we can also see that the *#ops* performed in the course of the metric-tree search, and the *#ops* required for computing similarity vectors for candidate sessions are independent of k. The results for I-TopK are depicted again by the blue horizontal line and consistently inferior to those of T-TopK. As for execution times, the average execution time for I-TopK is 47ms whereas T-TopK ranges from 22ms to 32ms. Graph omitted.

5.2 Synthetic data

The previous experiment demonstrates the utility of T-TopK, compared to I-TopK, on a repository that contains real analysis sessions. However, as the characteristics of the data may affect the algorithms' running times, we conducted a second set of experiments on synthetic data.

Data generation. Our goal here is to generate session repositories with different latent characteristics, and in particular control the repository size and level of similarity or queries and sessions. To generate a synthetic repository, we first built a query (metric-) space then built repository sessions by drawing queries from this place, as described next.

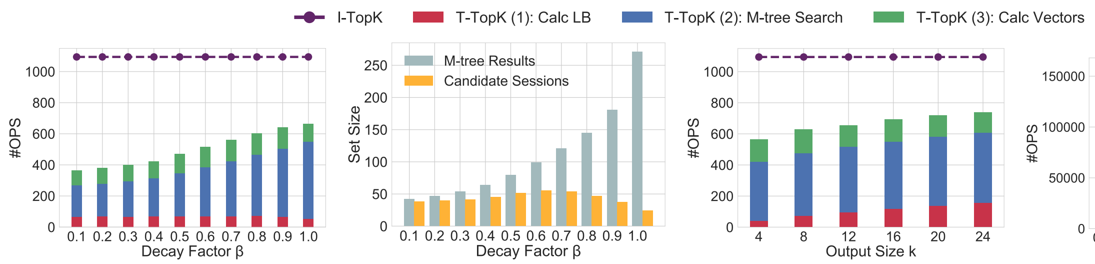
Generating the query space. Recall that our algorithms model queries as abstract objects in a given metric space. We represent here queries as points in an N-dimensional Euclidean space, with the Euclidean distance as the distance metric. To obtain values in $[0, 1]$ range, each element is restricted 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.

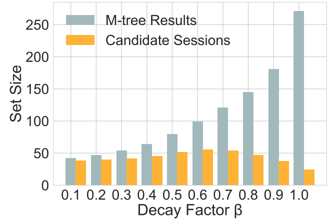
Using this construction method, we created a multitude of synthetic repositories, each with a different setting. Table 1 lists the construction parameters - the minimal, maximal, and default values used in the generation process.

Methodology and evaluation. We examine the effect of the above mentioned dataset parameters as well as the problem parameters (the gap penalty δ, the decay factor β and the output set size k), on the performance. In each experiment, we chose values for all parameters and generated a corresponding repository. From there we followed the same lines as in the real-data experiment, namely we ran the T-TopK and I-TopK algorithms with Euclidean distance as the query similarity measure.

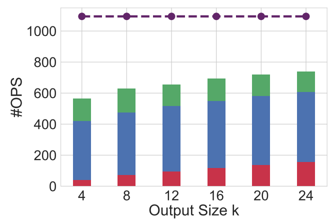
Results. Below we present selected results when varying a single parameter and setting the rest to a default value, as depicted in Table 1. When the *#ops* performed by I-TopK is not dependent on the values of the examined parameter, its performance is represented in the corresponding graph by a horizontal line. For T-TopK, we again measure the *#ops* in each of its three segments of computation. Due to the larger size of the repository, Segment 1, i.e. computing the lower bound, is negligible in all experiments, and thus hardly visible in the graphs. We start by examining the effect that the algorithm parameters have on the performance, this time using synthetic data. Then, we consider



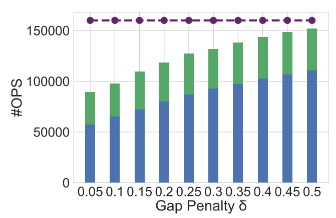
(a) R: Decay Factor β.



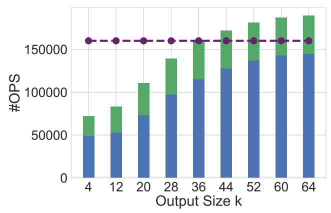
(b) R: M-tree Res./#Cands. (β).



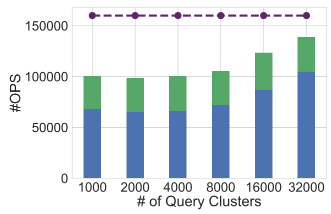
(c) R: Output Set Size k



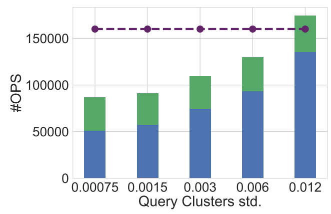
(d) S: Gap Penalty δ.



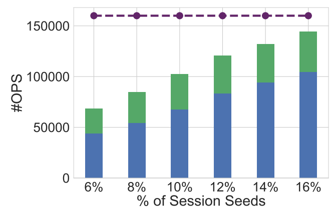
(e) S: Output Set Size k.



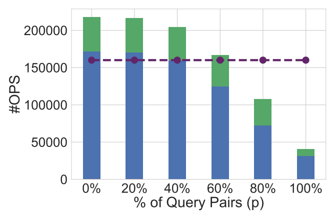
(f) S: # of Query Clusters



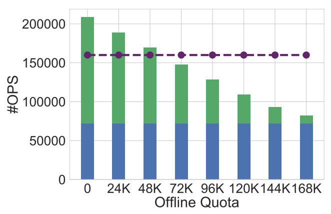
(g) S: Query Distance std.



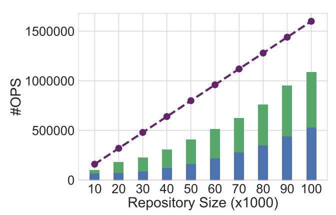
(h) S: % Seed Sessions



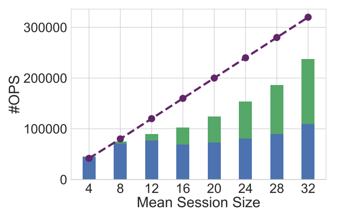
(i) S: % Query Pairs (p)



(j) S: Offline Quota



(k) S: Repository Size



(l) S: Mean Session len.

Figure 3: Parameters Effect on real life (R) and Synthetic (S) Datasets

the parameters determining the dataset properties.

Algorithm Parameters. The decay factor β yielded similar trends to the previous experiment, thus omitted. Next are the results for the gap penalty δ and the output size k.

Gap Penalty. Figure 3d displays the performance measured in *#ops* for T-TopK and I-TopK, for varying gap penalty value. As the results stay stable for $\delta > 0.5$ we omit them from the figure. We can see here some increase in the *#ops* when δ grows, particularly due to the metric-tree search. 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 query matching is almost always preferred over gapping, consequently further increase has no additional effect. As for execution times, I-TopK completes in 0.23 seconds whereas T-TopK ranges from 0.13 to 0.22 seconds (for δ values of 0.05 and 0.5 resp). Graph omitted.

Output set size k. As the synthetic repository is larger than the real-life dataset, we expanded the examination to larger k values. Figure 3e displays the performance for k varying from 1 to 64. Interestingly, compared to the results for the real-life 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 sets contains sessions with lower similarity scores, which in turn decreases the lower bound threshold inf^t. See that I-TopK is better suitable for larger k values (≥ 40), as it completes in 0.23 seconds regardless of k. However, recall that systems typically use values below 20. In this setting T-TopK takes less than 0.16 seconds.

Dataset Parameters. We start with the parameters defining the query space, then those of the sessions repository.

Query space parameters. 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 queries are retrieved from the metric-tree search. For instance, this can be seen in Figure 3f, which depicts the algorithms' performance when varying the number of clusters from $1K$ to $32K$ (resp., 0.2% to 20% of the overall number of queries), on a logarithmic scale. Here again T-TopK outperforms I-TopK. As expected, for T-TopK the *#ops* increases with the number of clusters, following an increase in the *#ops* performed in the metric-tree search. Correspondingly the running time of T-TopK varies from 0.14 ($1K$ clusters) to 0.2 seconds (32K clusters), comparing to 0.23 seconds obtained by I-TopK. A similar effect is observed when increasing the number of dimensions (figure omitted). The same principle also applies when increasing the standard deviation (std) of the distance between a query and its corresponding cluster center. See Figure 3g. In particular, for *std* ≥ 0.12 the lower bound is no longer effective, hence T-TopK takes 0.25 seconds, compared to I-TopK that takes 0.23 seconds.

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. Figure 3h depicts performance for varying number of seeds, from 6% to 16% of the sessions repository size. As expected, increasing the number of seeds induces an increase in *#ops* but even at 16% the performance is still better than that of I-TopK, with running times varying from 0.1 (6%) to 0.2 seconds (16%), compared to 0.23 seconds obtained by I-TopK.

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 3i, the performance of T-TopK improves as p increases, obtaining running times between 0.3 ($p = 0$) to 0.06 seconds ($p = 100\%$). For values under 60% I-TopK is more effective.

Repository size. Two parameters affect the repository size: the number of sessions, and the sessions' length. Figure 3l illustrates the performance as a function of the number of sessions in the repository. We see a linear increase in both algorithms, with T-TopK being more efficient than I-TopK by 41% on average. The running time for T-TopK ranges from 0.14 seconds for 10K sessions to 1.56 seconds for $100K$, compared to 0.23 to 2.29 seconds for I-TopK. A similar trend is measured when varying the mean session length, see Figure 3k. The only exception is that for very short sessions (average length 4) I-TopK is marginally more effective than T-TopK.

Offline Work. Finally, we measured how the expected idle-time affects the performance of T-TopK. As different query similarity functions take different time to compute, rather than varying the time interval we vary the number of similarity operations that can be computed (denoted offline quota). In Figure 3j the expected quota ranges from 0 to $168K$ operations (which translates to 0.24 seconds idle time), where the latter is typically sufficient to compute all missing vectors offline. As expected, the performance of I-TopK is invariant while that of T-TopK improves with the increase in the quota, as more missing similarity vectors are computed offline rather than during the top-k computation. When all missing vectors are computed offline, T-TopK takes only 0.12 seconds, compared to 0.23 seconds for I-TopK.

6. RELATED WORK

As explained in the Introduction, IDA recommender systems often use similar (sub)sessions to generate next-step recommendations [20, 16, 39, 17, 3]. In this work we employ a comprehensive similarity measure that was defined in [4], adapted to our context. The measure is based on the well known Smith-Waterman Algorithm [33] for sequence alignment. In the comparative study of [4] this measure was found to be most suitable, quality-wise, for the context of analysis sessions, compared to less comprehensive measures such as set-based (e.g. Jaccard, Dice) or edit-distance based measures [12]. Consequently, it has been adopted in several recent works [7, 3, 40, 43].

Local sequence alignment, and particularly the Smith-Waterman algorithm received considerable attention in the literature, shown to be extremely useful in the bioinformatics domain [26, 23, 36]. Thereby, a multitude of optimization frameworks are proposed for similarity search. Most solutions for similarity search (also known as approximate search) e.g. [19, 15, 38, 30] stem from the bioinformatics problem settings, typically DNA and protein sequencing. In this application domain, alignment is often performed between a query sequence and an extremely long *reference* string. Often in such solutions (see [24] for a survey), the large reference string (e.g. the human genome) is indexed, mostly by using a prefix/suffix tree or hash tables mapping small, common substrings to their location in the reference string. These solutions often assume a fixed small alphabet (e.g. nucleotides). In contrast, in our context the "alphabet" (namely the query space) is unbounded and there exist a distance metric between letters (queries). Incorporating ideas from genome sequencing into the IDA context is a challenging research direction.

Incremental similarity computation has been examined before in the context of string *edit distance.* Works e.g. [22, 18] suggest algorithms that when given strings A and B, and their *distance-matrix*, compute the distance of Ax and B (where x is an additional character) in $O(|A| + |B|)$. However, to the best of our knowledge, they focus on a single pair computation, rather than incremental similarity search in a large repository. We use similar principles for our incremental alignment computation (Section 3) but augment them with a novel threshold-based solution to support top-k search in large repositories. Similarity search in a repository has also been studied in the context of XML/graph/workflow databases [11, 41, 35]. These works mostly focus on structural similarity (rather than alignment), and do not support an underlying distance metric between objects as required for individual queries in the IDA context.

Last, efficient solutions have been suggested for top-k similarity search in non-metric spaces, considering complex/high-dimensional objects [1, 14, 8]. (Recall that while *query* similarity forms a metric space, the alignment-based *sessions* similarity does not.) Using these 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). Furthermore, they do not exploit the incremental nature of the search and thus at each step, all alignment computations are performed. As mentioned, we store individual queries in a metric tree [10], a popular efficient solution [32, 31, 5]. Our solution is modular and can in principle employ any alternative index mechanisms for searching a metric space (e.g.[29, 9, 25]).

7. CONCLUSION

In this work we present an efficient computational framework for the iterative retrieval of top-k similar sessions, a central component in many IDA recommender systems. We provide a simple generic model for describing IDA sessions, coupled with an alignment-based similarity notion. Using this model we devise effective algorithms that exploit the incremental nature of IDA sessions to speed up the top-k search. Our model and algorithms are applicable to a variety of analysis platforms, from the conventional SQL/OLAP, to modern tools such as Tableau and Splunk. We demonstrate the effectiveness of our solution via an extensive experimental evaluation, on both real-life and synthetic data.

In the previous section we mentioned multiple directions for future research, including the incorporation of ideas from bio-sequencing into the IDA context. Employing approximation techniques is another challenging future work.

8. REFERENCES

[1] M. R. Ackermann, J. Blömer, and C. Sohler. Clustering for metric and nonmetric distance measures. *ACM Transactions on Algorithms (TALG)*, 6(4):59, 2010.

[2] S. Aissi, M. S. Gouider, T. Sboui, and L. B. Said. Personalized recommendation of solap queries: theoretical framework and experimental evaluation. In *ACM Symposium on Applied Computing*, 2015.

[3] J. Aligon, E. Gallinucci, M. Golfarelli, P. Marcel, and S. Rizzi. A collaborative filtering approach for recommending olap sessions. *Decision Support Systems*, 69:20–30, 2015.

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

[5] A. Alpkocak, T. Danisman, and U. Tuba. A parallel similarity search in high dimensional metric space using m-tree. In *NATO Advanced Research Workshop on Cluster Computing*, pages 166–171. Springer, 2001.

[6] K. Aouiche, P.-E. Jouve, and J. Darmont. Clustering-based materialized view selection in data warehouses. In *East European Conference on Advances in Databases and Information Systems*, pages 81–95. Springer, 2006.

[7] 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.

[8] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *TPAMI*, 2013.

[9] C. C. M. Carélo, I. R. V. Pola, R. R. Ciferri, A. J. M. Traina, C. Traina Jr, and C. D. de Aguiar Ciferri. Slicing the metric space to provide quick indexing of complex data in the main memory. *Information Systems*, 36(1):79–98, 2011.

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

[11] S. Cohen and N. Or. A general algorithm for subtree similarity-search. In *ICDE*, 2014.

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

[13] M. Drosou and E. Pitoura. Ymaldb: exploring relational databases via result-driven recommendations. *The VLDB Journal*, 22(6), 2013.

[14] A. Eckhardt, T. Skopal, and P. Vojtáš. On fuzzy vs. metric similarity search in complex databases. In *International Conference on Flexible Query Answering Systems*, pages 64–75. Springer, 2009.

[15] R. C. Edgar. Search and clustering orders of magnitude faster than blast. *Bioinformatics*, 26(19):2460–2461, 2010.

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

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

[18] L. Hasan, Z. Al-Ars, and S. Vassiliadis. Hardware acceleration of sequence alignment algorithms-an overview. In *Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, pages 92–97. IEEE, 2007.

[19] W. J. Kent. Blatthe blast-like alignment tool. *Genome research*, 12(4):656–664, 2002.

[20] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu. Snipsuggest: Context-aware autocompletion for sql. *VLDB*, 2010.

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

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

[23] 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.

[24] H. Li and N. Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

[25] W. Lu, J. Hou, Y. Yan, M. Zhang, X. Du, and T. Moscibroda. Msql: efficient similarity search in metric spaces using sql. *The VLDB Journal*, 26(6):829–854, 2017.

[26] E. R. Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133–141, 2008.

[27] C. Miles. More like this: Query recommendations for sql. *Department of Computer Science & Engineering. University of Washington, Seattle, WA, USA*, 2011.

[28] T. Milo and A. Somech. React: Context-sensitive recommendations for data analysis. In *SIGMOD*, 2016.

[29] D. Novak and P. Zezula. M-chord: a scalable distributed similarity search structure. In *Proceedings of the 1st international conference on Scalable information systems*, page 19. ACM, 2006.

[30] P. Papapetrou, V. Athitsos, G. Kollios, and D. Gunopulos. Reference-based alignment in large sequence databases. *Proceedings of the VLDB Endowment*, 2(1):205–216, 2009.

[31] T. Skopal. Pivoting m-tree: A metric access method for efficient similarity search. In *DATESO*, volume 4, pages 27–37. Citeseer, 2004.

[32] T. Skopal and B. Bustos. On nonmetric similarity search problems in complex domains. *ACM Computing Surveys (CSUR)*, 43(4):34, 2011.

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

[34] L. Spitzner. The honeynet project: Trapping the hackers. *IEEE S&P*, 2003.

[35] J. Starlinger, S. Cohen-Boulakia, S. Khanna, S. B. Davidson, and U. Leser. Effective and efficient similarity search in scientific workflow repositories. *Future Generation Computer Systems*, 56:584–594, 2016.

[36] T. A. Tatusova and T. L. Madden. Blast 2 sequences, a new tool for comparing protein and nucleotide sequences. *FEMS microbiology letters*, 174(2):247–250, 1999.

[37] R. Uribe-Paredes, P. Valero-Lara, E. Arias, J. L. Sánchez, and D. Cazorla. Similarity search implementations for multi-core and many-core processors. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 656–663. IEEE, 2011.

[38] S. Wandelt, J. Starlinger, M. Bux, and U. Leser. Rcsi: Scalable similarity search in thousand (s) of genomes. *Proceedings of the VLDB Endowment*, 6(13):1534–1545, 2013.

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

[40] 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.

[41] Y. Yuan, G. Wang, L. Chen, and H. Wang. Efficient subgraph similarity search on large probabilistic graph databases. *Proceedings of the VLDB Endowment*, 5(9):800–811, 2012.

[42] P. Zezula, P. Savino, F. Rabitti, G. Amato, and P. Ciaccia. Processing m-trees with parallel resources. In *Research Issues In Data Engineering, 1998.'Continuous-Media Databases and Applications'. Proceedings., Eighth International Workshop on*, pages 147–154. IEEE, 1998.

[43] Z. Zolaktaf. Facilitating user interaction with data. 2017.

APPENDIX

A. ADDITIONAL PROOFS

In the following, we provide additional proofs complementing Section 3 and Section 4.

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

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

PROOF SKETCH. We prove for the case that the lastly added query of u_t is aligned to some query $s[j]$ (Following similar lines, one can prove for the cases where a gap penalty is taken).

By Equation 2 we get:

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

Due to Proposition 3.2 and the latter transition, we get:

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

Thus, $\vec{Sim}(u_t, s)[j] = \vec{Sim}(u_{t-1}, s)[j-1]\beta^2 + \sigma(u_t[t], s[j])$. Last, it is left to prove that: $\vec{Sim}(u_{t-1}, s)[j-1]\beta^2 + \sigma(u_t[t], s[j]) \geq max(\vec{Sim}(u_t, s)[j-1]\beta - \delta, \vec{Sim}(u_{t-1}, s)[j]\beta - \delta)$ then the theorem holds. $\square$

PROPOSITION A.2. *$min\text{–}score(top_k(u_{t-1}, S))\beta - \delta \leq inf^t \leq min-score(top_k(u_t, S))$*

PROOF (SKETCH). The proof of the left hand side inequality follows immediately from the observation that for any $S' \subseteq S$, $min\text{–}score(top_k(u_{t-1}, S))\beta - \delta \leq min\text{–}score(top_k(u_t, S')) \leq min\text{–}score(top_k(u_t, S))$. 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 $min-score(top_k(u_{t-1}, S))\beta - \delta$ $\square$

PROPOSITION A.3. *For a session $s \in S \setminus S_{top}^{t-1}$, if when computing $Sim(u_t, s)$ the query q_t is not matched with any of s's queries, then for every prefix s_j of s, $Sim(u_t, s_j) \leq inf^t$*

PROOF. 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 Theorem 3.4. 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 $Sim(u_t, s_j) \leq min-score(top_k(u_{t-1}, S))\beta - \delta \leq inf^t$ 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. $\square$

PROPOSITION A.4. *For every session s, $\widehat{sup}^t(s) \leq min-score(top_k(u_{t-1}, S))\beta^2 + max_{q \in s}\{\sigma(q_t, q)\}$*

PROOF SKETCH. Let s_m be the prefix of s with the maximal similarity to u_t. Recall that following Proposition 4.4 we assume that q_t aligns with a query in s, and using Theorem 3.4, 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 $Sim(u_{t-1}, s_{m-1}) \leq min-score(top_k(u_{t-1}, S))$, and $\sigma(q_t, s_m[m]) \leq max_{q \in s}\{\sigma(q_t, q)\}$, thus the proposition holds. $\square$

PROPOSITION A.5.

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

PROOF. 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:

$$\widehat{sup}^{t+1}(s) \leq \widehat{sup}^t(s)\beta^2 + max_{q \in s}\{\sigma(u_{t+1}[t+1], q)\} \quad (3)$$

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 (3) 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 $t + 1$ $\square$