Durable Top-K Instant-Stamped Temporal Records with User-Specified Scoring Functions (Technical Report Version)

Junyang Gao[§]
Google Inc.

Stavros Sintos[§]
University of Chicago

Pankaj K. Agarwal
Duke University

Jun Yang
Duke University

Abstract—A way of finding interesting or exceptional records from instant-stamped temporal data is to consider their "durability," or, intuitively speaking, how well they compare with other records that arrived earlier or later, and how long they retain their supremacy. For example, people are naturally fascinated by claims with long durability, such as: *"On January 22, 2006, Kobe Bryant dropped 81 points against Toronto Raptors. Since then, this scoring record has yet to be broken."* In general, given a sequence of instant-stamped records, suppose that we can rank them by a user-specified scoring function f, which may consider multiple attributes of a record to compute a single score for ranking. This paper studies *durable top-k queries*, which find records whose scores were within top-k among those records within a "durability window" of given length, e.g., a 10-year window starting/ending at the timestamp of the record. The parameter k, the length of the durability window, and parameters of the scoring function (which capture user preference) can all be given at the query time. We illustrate why this problem formulation yields more meaningful answers in some practical situations than other similar types of queries considered previously. We propose new algorithms for solving this problem, and provide a comprehensive theoretical analysis on the complexities of the problem itself and of our algorithms. Our algorithms vastly outperform various baselines (by up to two orders of magnitude on real and synthetic datasets).

I. INTRODUCTION

Instant-stamped temporal data consists of a sequence of records, each timestamped by a time instant which we call the arrival time, and ordered by the arrival time. Such data is ubiquitous in a rich variety of domains; i.e., sports statistics, weather measurement, network traffic logs and e-commerce transactions. A way of finding interesting or unusual records from such data is to consider their "durability," or, intuitively speaking, how well they compare with other records (i.e., records that arrive earlier or later) and how long they retain the supremacy. For example, consider the performance record: "On January 22, 2006, Kobe Bryant scored 81 points against Toronto Raptors." While impressive by itself, this statement can be boosted by adding some temporal context: "At that time, this record was the top-1 scoring performance *in the past 45 years of NBA history*." Naturally, the further back we can extend the "durability" (while the record still remains top), the more convincing the statement becomes. We can extend durability forward in time as well: "Since 2006, Kobe's
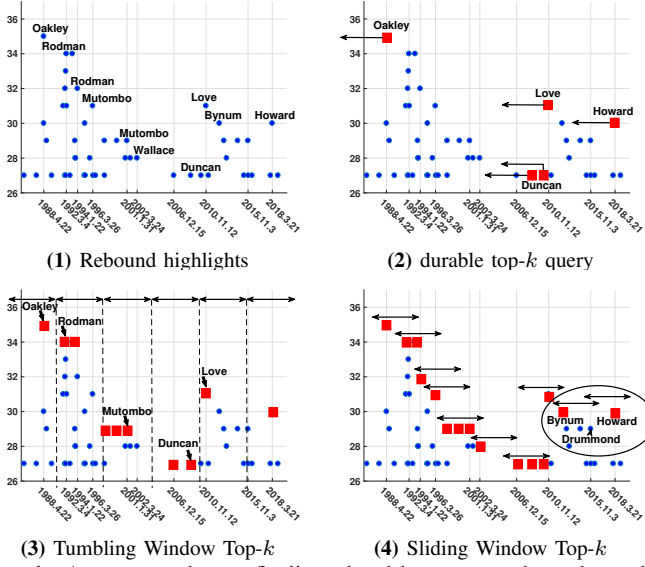
81 points scoring performance has yet to be broken as of today." The notion of durability is widely used in media and marketing, because people are naturally attracted by those events that "stood the test of time." Such analysis of durability is a useful part of the toolbox for anybody who works with historical data, and can be particularly helpful to journalists and marketers in identifying newsworthy facts and communicating their impressiveness to the public. Because temporal data can accumulate to very large sizes (especially for granular data such as weather or network statistics), and because users often want to find durable records with respect to different ranking criteria quickly, we need to answer durable top-k queries efficiently.

In this paper, we consider *durable top-k queries* for finding instant-stamped records that stand out in comparison to others within a surrounding time window. In general, each record may have multiple attributes (besides the timestamp) whose values are relevant to ranking these records. We assume that there is a user-specified scoring function f that takes a record as input, potentially considers its multiple attributes, and computes a single numeric score used for ranking. Intuitively, a durable top-k query returns, given a time duration τ, records that are within top k during a τ-length time window anchored relative to the arrival time of the record. How the window should be positioned relative to the arrival time depends on the application; our solution only stipulates that the relative positioning is done consistently across all records. In practice, we observe most statements in media involving durability either *ends* the window at the arrival time of the record (i.e., *looking back* into the past) or *begins* the window at the arrival time of the record (i.e., *looking ahead* into the future). Generally speaking, each record returned by our durable top-k corresponds to a statement about the record that highlights the durability of its supremacy.

Note that there are different ways for capturing the notion of durability in queries, including some types that have been studied in the past. Different application scenarios may call for different semantics. To understand why our definition of durable top-k queries may be more appropriate than others in some scenarios, we examine the alternatives with a simple concrete example.

Example I.1. *Suppose we are interested in finding exceptional*

(1) Rebound highlights

(2) durable top-k query

(3) Tumbling Window Top-k

(4) Sliding Window Top-k

Fig. 1: A case study on finding durable noteworthy rebound performances in NBA history. Red squares highlight results returned by different queries, and line segments represent the durability time window.

rebounds performances (by individual players in individual games) in NBA history—particularly, those that stood out as the top record (or tying for the top record) in a 5-year time span. Figure 1.(1) plots all relevant records (i.e., no fewer than 27 rebounds by a single player in a single game) in entire NBA history. We consider the following three queries to accomplish our task; the latter two have been widely studied in the stream processing and top-k query processing literature. Note that in this example $k = 1$.

- **Durable top-k (our query):** This is the query that we propose. For each record, we look back in a 5-year window ending at the timestamp of the record, and check whether the record has the top score among all records within this window. Figure 1.(2) highlights the records (red squares) returned by our query; for each result record, we also show its 5-year durability window as a line segment ending at the record for which it remains on the top.

- **Tumbling-window top-k:** This query first partitions the timeline into a series of non-overlapping, fixed-sized (5-year) windows, and then returns the top record within each time window. The placement of the windows is up to the user and can affect results. Results for one particular placement of the windows are shown in Figure 1.(3).

- **Sliding-window top-k:** This query slides a 5-year window along the timeline, and returns the top record for each position of the sliding window. Figure 1.(4) highlights a few representative sliding windows, as well as the top records during these windows.

All these queries are able to uncover some meaningful durable top records; i.e., for any data record (X, Y, Z) marked as a red square in Figure 1, we can claim "player X grabbed

Y rebounds in a game on date Z, which is the best in **some** 5-year span." First, the durability aspect adds to the impressiveness of the statement. Second, the combination of durability and ranking helps reveal interesting records that would otherwise be ignored if we simply filter the records by a high absolute value. For instance, all three queries find (Duncan, 27, 2009) as a durable top-1 record. While this record may not seem impressive by number alone, it was indeed the top-1 from 2002 to 2010. This is an interesting observation, as it reflects a trend (relatively low rebounds of all players) during that era of NBA.

However, there are also notable differences.

- **Tumbling-window vs. our query:** The general observation is that the results of tumbling-window are highly sensitive to the choice of window placement. In Figure 1.(3), tumbling-window picks (Mutombo, 29, 2001) and the other two performances with 29 rebounds as they were the best ones during 2000-2005, but there were more impressive performances right before them, unfortunately leaving the impression that they stood out only because the windows were cherry-picked. Furthermore, if we choose to place all windows slightly to the right such that the last window ends with the most recent arrival time, (Rodman, 34, 1992) will be eliminated by (Oakley, 35, 1988), and (Duncan, 27, 2009) will be overlooked since it is shadowed by (Love, 31, 2010). Overall, because of high sensitivity to window boundaries, tumbling-window runs the risk of omitting important records as they happen to be overshadowed by some other records in the same window, and picking less interesting records as they happen to be the top ones in that specific window.

- **Sliding-window vs. our query:** Sliding-window is not susceptible to window placement, but it effectively considers all possible window placements, and it returns the union of all top records for each such placement. This approach leads to possibly many records that are not as meaningful in practice. In Figure 1.(4), sliding-window apparently returns overwhelmingly more results compared to our query, which makes it less applicable to mining most noteworthy records. Even more unnatural is the fact that as we slide the window along the timeline, a record can come in and out of the result; i.e., there is no continuity. To illustrate, suppose we are interested in durable top-2 records with 5-year windows, and let us focus on Drummond's 29 rebounds performance on 2015.11.3 (highlighted in Figure 1.(4)). It is surrounded by two top performance (Howard, 30, 2018) and (Bynum, 30, 2013). Sliding-window will return this record when the window is positioned at 2014-2019, but not when positioned at 2013-2018; however, the record will be returned again when the window moves to 2012-2017. Such discontinuity makes the results rather unnatural to interpret.

In comparison, our query does not have the issue of sensitivity to window placement or that of difficulty of interpretation, because we assess each record in a 5-year window that leads up to its own timestamp. Thus, our query result records can be consistently interpreted as having durability "within the past 5 years" and clearly communicated to the audience. The

results from the other two queries would be qualified with rather specific durability windows,[1] which may be perceived as cherry-picking. In general, we argue that consistency and simplicity of our query make it more applicable to journalists, marketers, and data enthusiasts alike who seek result that are easily explainable to the public.

In comparison, our query does not have the issue of sensitivity to window placement or that of difficulty of interpretation, because we assess each record in a 5-year window that leads up to its own timestamp. Thus, our query result records can be consistently interpreted as having durability "within the past 5 years" and clearly communicated to the audience. The results from the other two queries would be qualified with rather specific durability windows, which may be perceived as cherry-picking.

Although the above example ranks records by a single attribute, its argument can be extended to the general case where records are ranked by a user-specified scoring function that combines multiple attribute values into a single score.

Besides sports, durable top-k queries have applications across many other domains. For instance, Wikipedia states that "In late January 2019, an extreme cold wave hit the Midwestern United States, and brought the coldest temperatures in the past 20 years to most locations in the affected region, including some all-time record lows." This statement stems from a simple durable top-k query over historical weather data, and allows the Wikipedia article to convey the severity of event effectively. As an example involving more complex ranking, cybersecurity analysts rely on network traffic log to identify unusual and potentially malicious intrusions. With a appropriately defined scoring function that combines multiple features of a session, such as duration, volume of data transfer, number of login attempts, and number of servers accessed, a durable top-k query can quickly help identify unusual traffic (relative to others around the same time) for further investigation. As another example, a financial broker may accompany a recommendation with a statement "The price-to-earnings ratio (P/E) of this stock last Friday was among the top 5 P/E's within its section for more than 30 days," which is also a durable top-k query. In sum, the efficiency of durable top-k queries makes them suitable for using large volumes of historical efficiently to drive insights or identify leads for further investigation; the conceptual simplicity of these queries also make them particular attractive for explaining insights and communicating them effectively to the public.

Contributions. Our contributions are as follows:
- We propose to find "interesting" records from large instant-stamped temporal datasets using durable top-k queries. Compared with other query types related to durability, our query produces results that are more robust (i.e., less sensi-

tive to window placement than tumbling-window) and more meaningful (i.e., easier to interpret than sliding-window).
- We propose a suite of solutions based on two approaches that process "promising" records in different prioritization orders. We provide a comprehensive theoretical analysis on complexities of the problem and of our proposed solutions.
- Our solutions are general and flexible. They do not dictate any specific scoring function f, but instead assume a well-defined building block for answering top-k queries using f, which can be "plugged into" our solutions and analysis. We give some concrete example of f and the building block in later sections. In particular, f can be further parameterized according to user preference; these parameters, along with k, τ and I (the overall temporal range of history of interest), can be specified at query time, making our solutions flexible and suitable for scenarios where users may explore parameter setting at run-time, interactively or automatically.
- We show that the query time complexity of our algorithms is proportional to $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$ in the worst case, where $|S|$ is the answer size. Furthermore, we prove that the expected answer size of a durable top-k query $|S|$ is $O(k\lceil\frac{|I|}{\tau}\rceil)$ under the random permutation model (where the data values can be arbitrarily chosen by an adversary but arrival order is random); this result implies that the expected query time of our algorithms in practice is linear in the output size.

Paper Overview. In a nutshell, our proposed algorithms 1) visit promising records in some manner, and 2) check the durability (with respect to a top-k query) for each record we visit. Techniques for improvement mostly focus on how to efficiently identify candidate records and eventually reduce the total number of durability checks in the second step. Our proposed algorithms come in two flavors: time-prioritized and score-prioritized, introduced in Section III and Section IV, respectively. The ***time-prioritized*** solution traverses and finds candidate records sequentially along the timeline, while the ***score-prioritized*** solution greedily chooses unvisited candidates with the maximum score (with respect to f). Though in different manners, we show in later sections that these two solutions actually equivalently reduce and bound the size of candidate records (or, the number of durability checks). More interestingly, in Section V, we further demonstrate that the bound is proportional to the answer size of a durable top-k query, which means our algorithms run faster when the query is more selective, e.g., with smaller k or longer durability τ. Section VI experimentally evaluates our proposed solutions, including implementations inside a database system. Section VII reviews related work and Section VIII concludes.

II. PROBLEM STATEMENT AND PRELIMINARIES

Problem Statement. Consider a dataset P with n records, where each record $p \in P$ has d real-valued attributes and is represented as a point $(p.x_1, p.x_2, \ldots, p.x_d) \in \mathbb{R}^d$. For simplicity, we consider a discrete time domain of interest $\mathbb{T} = \{1, 2, \ldots, n\}$, and let $p.t \in \mathbb{T}$ denote the *arrival time* of p. All records in P are organized by increasing order of their

---

[1] A related question is whether we can post-process the results of the sliding-window query to obtain the results to our query; e.g., filtering those result records in Figure 1.(4) to get those in Figure 1.(2). Unfortunately, such an approach, which we consider as one of the baselines in our experiments, is prohibitively slow on large datasets, as we shall show in later sections.

TABLE I: Table of notation

$\mathbb{T}$	Time domain
$p.t$	Arrival time of p
f	Scoring function
k	Parameter of Top-k query
$\pi_{\leq k}([t_1, t_2])$	Top-k records in time interval $[t_1, t_2]$
I	Query interval
τ	Durability duration
u	Query vector
$s(n), q(n)$	Space, query time of top-k index

arrival time. Given a non-empty time window $W : [t_1, t_2] \subseteq \mathbb{T}$, let $P(W)$ denote the set of records that arrive between t_1 and t_2; i.e., $P(W) = \{p \in P \mid t_1 \leq p.t \leq t_2\}$.

Assume a user-specified scoring function maps each record p to a real-valued score, $f : \mathbb{R}^d \to \mathbb{R}$. Given a time window $W = [t_1, t_2]$, a *top-k query* $Q(k, W)$ asks for the k records from $P(W)$ with the highest scores with respect to f. Let $\pi_{\leq k}([t_1, t_2])$ denote the result of $Q(k, W)$; i.e., for $\forall p \in \pi_{\leq k}([t_1, t_2])$, there are no more than $k - 1$ records $q \in P([t_1, t_2])$ with $f(q) > f(p)$.

For simplicity of exposition, we consider durability windows ending at the arrival time of each record (i.e., the "looking-back" version), but our solution can be extended to the general case where the windows are anchored consistently relative to the arrival times (including the "looking-ahead" version). We say a record p is τ-durable[2] if $p \in \pi_{\leq k}([p.t - \tau, p.t])$. That is, p remains in the top-k for τ time during $[p.t - \tau, p.t]$. We are interested in finding records with long durability. Note that if a record p is τ-durable, then it is also τ'-durable for $\tau' \leq \tau$. We are interested in finding records with "long enough" durability, i.e., durability at least τ. Given a query interval I and a durability threshold $\tau \in [1, |\mathbb{T}|]$, a *durable top-$k$ query*, denoted $\mathsf{DurTop}(k, I, \tau)$, returns the set of τ-durable records that arrive during I; i.e., $\mathsf{DurTop}(k, I, \tau) = \{p \in P(I) \mid p \in \pi_{\leq k}([p.t - \tau, p.t])\}$. For a record $p \in \mathsf{DurTop}(k, I, \tau)$ we can also ask what is the maximum duration that it remains in the top-k. Table I summarized our notations.

Scoring Function and Top-k Query Building Block. As discussed earlier, our proposed algorithms and complexity analyses are applicable to any user-specified scoring function f as long as there exists a "building block" that can answer basic (non-durable) top-k queries under f. This building block can be a "black box": the novelty and major contribution of our algorithms come from its ability to reduce and bound the number of invocations of the building block, totally independent of how the building block operates itself. Of course, the overall algorithm complexity still depends on the efficiency of the building block. For a function f, we consider that an index of size $O(s(n))$ can be constructed in $O(u(n))$ time that answers top-k queries with respect to f in $O(q(n) + k)$ time, where n is the data size and $s(\cdot), u(\cdot), q(\cdot)$ are functions of n.

In this paper, we are more interested in top-k queries on a subset of data specified by a time window W given at query time; i,e., computing $Q(k, W)$ that reports the k records in $P(W)$ with the highest scores with respect to f. With a slight

care, the top-k query building block can be used to solve this problem by paying a logarithmic factor in index size, query time and construction time. That is, for a function f we can construct an index of size $O(s(n) \log n)$ in $O(u(n) \log n)$ time so that for given k, W, $Q(k, W)$ can be computed in $O((q(n) + k) \log n)$ time. If the top-k building block supports updates (insertion/deletion of an item) in $O(\alpha(n))$ time, our range top-k index also supports updates in $O(\alpha(n) \log n)$ time.

Here, we give some concrete examples of f that are widely used in real-life applications, for which efficient top-k query building blocks exist. Consider the following class of scoring functions parameterized by $\mathbf{u}$, which captures user preference:
- *linear*: $f_{\mathbf{u}}(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$,
- *linear combination of monotone scoring functions*: $f_{\mathbf{u}}(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot h(p.x_i)$, where h is a monotone function; i.e., $h(\cdot) = \log(\cdot)$,
- *cosine*: $f_{\mathbf{u}}(p) = \frac{1}{|p||\mathbf{u}|} \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$,

where $\mathbf{u}$ is a real-valued preference vector and $f_{\mathbf{u}}$ denotes that the scoring function f is parameterized by $\mathbf{u}$. We refer to this class of functions as *preference functions*. Top-k queries using such class of scoring functions (preferably in the above three forms) have been well studied over the past decades both in computational geometry [1]–[6] and databases [7]–[10]. For example, for preference functions above, there is an index with $u(n) = O(n)$, $s(n) = O(n)$, and $q(n) = O(n^{1-1/\lfloor d/2 \rfloor})$, skipping $\mathrm{polylog}(n)$ factors. Using the results in [5], updates can also be supported in $\alpha(n) = O(\mathrm{polylog}(n))$ time.

As mentioned above, users can replace the scoring block with other functions (i.e., non-linear or non-monotone). The centerpiece of our algorithm and analysis, which bounds the *number of invocations* of the top-k query building block, remains unchanged. But in that case, the complexity of the building block will affect the overall complexity bound. We choose these functions because 1) they are widely used in real-life applications that require ranking and 2) they are both linear and monotone, so preference top-k can be efficiently answered (using the same index).

Sliding-Windows and Baseline Solution. Recall from the discussion in Example I.1 (Figures 1-(2) and 1-(4)) that there is a connection between our problem and the sliding-window version, which has been well studied [11]–[13]. Indeed, one of our baseline solution is adopted from [11] with incremental top-k maintenance over sliding windows[3]. However, the standard sliding-window technique is more suitable for data streams, where incoming data must be scanned linearly anyway. Instead, our query analyzes historical data. The linear complexity of sliding windows becomes infeasible especially when dealing with large datasets. The limitation hence motivates our solutions in later sections. Experimental results demonstrate our algorithms' significant efficiency gain (up to 2 orders of magnitude) over sliding-window baselines.

Duration of durable top-k records. When an algorithm finds a record p in $\mathsf{DurTop}(k, I, \tau)$, we can also get the maxi-

---

[2]If τ is obvious from the context, we drop τ from the definition, i.e., we say that a record is durable.

[3]In particular, the idea of Skyband Maintanence Algorithm (SMA) to reduce the number of top-k re-computations from scratches.

mum duration (in history) that it remains in the top-k. We do it by running a binary search with respect to the arrival times of the records back in history. For each step of the binary search we ask a top-k query to check if p is still in the top-k records. The correctness follows from the observation that if a record is τ'-durable then it is also τ-durable for any $\tau < \tau'$. The binary search has $O(\log n)$ steps and each top-k query takes $O(q(n))$ time. For all records in $|\mathsf{DurTop}(k, I, \tau)|$ this procedure takes $O(|\mathsf{DurTop}(k, I, \tau)| \cdot q(n) \log n)$ time. Notice that this procedure is independent of the algorithm we use to find the τ-durable records in I, so it can be applied in the end of all the algorithms we propose in the next sections (without increasing their total running time).

III. Time-Prioritized Approach

The time-prioritized approach is straightforward: we visit records in time order and check their durability. We start with a baseline approach (Section III-A) and propose an improved version (Section III-B) using the observation that we can skip many unpromising records in practice. What is more interesting is how this simple improvement leads to provably substantial reduction in complexity (Section III-C).

A. Time-Baseline Algorithm

We start with a baseline solution, referred to as *Time-Baseline* or *T-Base*. T-Base shares the same spirit as the solution proposed in [11], where authors studied the problem on how to continuously monitor top-k queries over the most recent data in a streaming setting. The main idea is to incrementally maintain the top-k set over continuous sliding windows. We start with the right endpoint of query interval, and sequentially slide a τ-length window backwards along the timeline. For each sliding window $[t - \tau, t]$, we need the top-k result to check whether the record (arriving at time t) is τ-durable. With two adjacent windows $W_1 = [t - \tau, t]$ and $W_2 = [t - \tau - 1, t - 1]$, top-$k$ results could be updated incrementally, if the expired record (e.g., $P[t]$) is not a top-k on W_1. Otherwise, we need to compute the top-k on window W_2 from scratch to guarantee correctness. The procedure repeats until we visit all records in the query interval I.

Next, we analyze the query time complexity of T-Base. There are only two types of records: durable or non-durable. After visiting each durable record, we need to issue a top-k query. After visiting each non-durable record, we only need to incrementally update the current top-k set with new incoming record in $O(\log k)$ time. Assuming a top-k query can be answered in $O((q(n) + k) \log n)$ time, then T-Base runs in $O\big(|S|(q(n) + k) \log n + n \log k)\big)$, where $|S|$ is the answer size. This algorithm takes super-linear time (on the number of records in the query interval). Next, we show a solution with sub-linear query time.

B. Time-Hop Algorithm

It is not hard to see that the durable top-k query can be viewed as an *offline* version of the top-k query in the sliding-window streaming model. Hence, the baseline algorithm introduced above does not best serve our needs. Since the entire
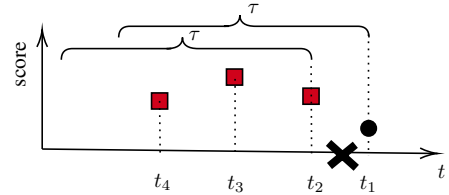

Fig. 2: Data skipping in Time-Hop Algorithm.

Algorithm 1: T-Hop (k, I, τ)

Input: P, k, τ, and $I : [t_1, t_2]$.
Output: $\mathsf{DurTop}(k, I, \tau)$
1 Initialize answer set: $S \leftarrow \varnothing$, top-k set: $\pi_{\leq k} \leftarrow \varnothing$;
2 $t_{curr} \leftarrow t_2$;
3 **while** $t_{curr} >= t_1$ **do**
4    $\quad \pi_{\leq k} \leftarrow Q(k, [t_{curr} - \tau, t_{curr}])$;
5    $\quad$ **if** $P[t_{curr}] \in \pi_{\leq k}$ **then**
6    $\quad\quad S \leftarrow S \cup P[t_{curr}]$;
7    $\quad\quad t_{curr} \leftarrow t_{curr} - 1$;
8    $\quad$ **else**
9    $\quad\quad t_{curr} \leftarrow$ most recent arrival time of records in $\pi_{\leq k}$;

10 **return** S;

---

data is available in advance, the manner of continuous sliding window wastes too much time on those non-durable records. After all, a meaningful durable top-k query should be selective.

Before describing the algorithm, we illustrate the main idea using an example for $k = 3$, shown in Figure 2. By running a top-3 query $Q(3, [t_1 - \tau, t_1])$, consider the record p arriving at t_1 (black circle) is not τ-durable; i.e., $p \notin \pi_{\leq 3}([t_1 - \tau, t_1])$. We know the current top-3 set contains records (red squares) that arrive at t_4, t_3 and t_2. Then, no records arriving between t_2 and t_1 would be τ-durable and we can safely hop from t_1 to t_2. This simple and useful observation simplifies the query procedure, and allows larger strides for sliding windows.
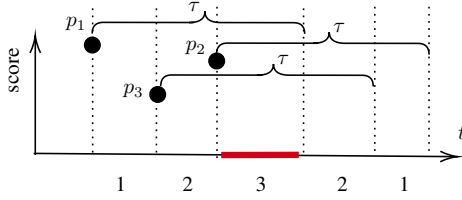
Now, we present our algorithm *Time-Hop* (T-Hop) (the pseudocode can be found in Algorithm 1). For each record we visit with timestamp t_i, we run a top-k query in $[t_i - \tau, t_i]$ (Line 4). If the record is not durable, we slide the window back to the *most recent* arrival time of records, say t_j, in the current top-k set (Line 9), skipping the non-durable records between t_j and t_i. Otherwise, if a durable record is found, we slide the window backwards by 1 (Line 7) as usual. Note that if we adopt the look-ahead version of durability, we just need to reverse the traversal order (and time-hopping) on timeline as well.

C. Complexity Analysis of T-Hop

For the Time-Hop algorithm, the time complexity purely depends on the number of top-k queries called in the query procedure. We provide a worst-case guarantee on the number of top-k queries performed, as shown by the lemma below (See Appendix B for full proofs).

Lemma 1. *The total number of top-k queries performed by the Time-Hop algorithm is $O\big(|S| + k\lceil \frac{|I|}{\tau} \rceil \big)$.*

Fig. 3: Blocking mechanism in score-prioritized approach

Proof (Sketch). For each record we visit in T-Hop, a top-k query is called for a durability check. If the record is not τ-durable, we refer it to as a *false check*. Otherwise, we add it to the answer set. Hence, we only need to bound the total number of false checks. We decompose the total number of false checks into a set of disjoint τ-length windows, and derive an upper bound of false checks that happen in such a window.

In particular, let ρ be a window of length τ and let S_ρ be the τ-durable records in ρ. We divide the false checks in ρ into two types. If a false check appears immediately after a τ-durable record (found by the algorithm) then this is a type-1 false check. Otherwise it is a type-2 false check. From the definition, the number of type-1 false checks in ρ is $O(S_\rho)$. Furthermore, we show that after finding i type-2 false checks in ρ, a top-k query (that is called for durability check) can only find $k - i$ records in ρ. In that way we show that the number of type-2 false checks is $O(k)$.

Given a query interval I, there are at most $\left\lceil \frac{|I|}{\tau} \right\rceil$ disjoint τ-length sub-intervals. We conclude that the number of top-k queries is $O\left(|S| + k\left\lceil \frac{|I|}{\tau} \right\rceil\right)$. ∎

Overall, with an efficient top-k module, T-Hop answers a durable top-k query $\mathsf{DurTop}(k, I, \tau)$ in $O\left(\left(|S| + k\left\lceil \frac{|I|}{\tau} \right\rceil\right)(q(n) + k)\log n\right)$ time. Compared to T-Base, T-Hop runs in sublinear query time (assuming that the ratio $\left\lceil \frac{|I|}{\tau} \right\rceil$ is not arbitrarily large), i.e., the running time does not have a linear dependency on the number of records in I. Our experimental results in Section VI suggests that T-Hop is one to two orders of magnitude faster than T-Base in practice. Furthermore, we recall that our index can be implemented with near linear size and polylogarithmic update time for preference queries.

Notice that the number of top-k queries performed by T-Hop depends on $|S|$ and $k\left\lceil \frac{|I|}{\tau} \right\rceil$. Ideally, we would like to argue that the number of top-k queries is $O(|S|)$. In theory, the term $k\left\lceil \frac{|I|}{\tau} \right\rceil$ can be arbitrarily large comparing to $|S|$. In Section V-A we study the expected size of S in a random permutation model where a set of n scores, chosen by an adversary, are assigned randomly to the records. In such a case we show that the expected size of S is roughly $O(k\left\lceil \frac{|I|}{\tau} \right\rceil)$, meaning that in practice we expect that the number of top-k queries we execute are asymptotically equal to $|S|$.

IV. Score-Prioritized Approach

One weakness of time-prioritized approach is that it does not pay much attention to scores and simply visit records sequentially along the timeline (with hops). Though Lemma 1 shows that T-Hop visits $O(|S| + k\left\lceil \frac{|I|}{\tau} \right\rceil)$ records in the worst case, it still potentially visits many low-score and non-durable records and ask more top-k queries. In contrast, the score-

prioritized approach visits candidate records in descending order of their scores because records with high scores have a higher chance of being durable top-k records. Furthermore, these high-score records can also serve as a benchmark for future records, enabling a "blocking mechanism" to prune candidates.

Before describing the algorithms, we illustrate the main idea using an example shown in Figure 3. Suppose we answer a durable top-3 query with τ by visiting records in descending order of their scores: p_1, p_2 and p_3, and all three records are durable ones. p_1 has the highest score in the entire query interval, any record that lies in the τ-length time interval $[p_1.t, p_1.t + \tau]$ will be dominated by p_1, which we refer to as being "blocked" by p_1. Similarly, p_2 (the second highest score) and p_3 (the third highest score) also block a τ-length interval starting from their arrival times. The time axis is partitioned into intervals by endpoints of all blocking intervals. In Figure 3, the number under each interval shows how many records block this interval. Notice the bold red interval, where any record in this interval lies in three blocking intervals after processing p_1, p_2 and p_3. Since there are already three records with higher score than any record in this interval, it can not have any τ-durable top-3 record, and we can safely remove this time interval from consideration. As we continue adding blocking intervals, eventually every remaining record in the query interval will be blocked by at least three blocking intervals. The algorithm can now stop because no more durable top records can be found. The procedure is straightforwardly applicable to look-ahead version of durability, by simply reversing the direction of blocking intervals.

We describe three algorithms in the following sections. They differ on how the high-score records are found and how the blocking intervals are maintained.

A. Score-Baseline Algorithm

We start with a baseline method (S-Base) of score-prioritized approach, which sorts records in the query interval in descending order of their scores. Given k, τ and a query interval $[t_1, t_2]$: (1) Sort all records in time interval $[t_1 - \tau, t_2]$ in descending order of scores. (2) For each record p in sorted order: If $p.t \in [t_1, t_2]$ and p lies in less than k blocking intervals, add p to answer set; Otherwise, continue. In any case, add a blocking interval $[p.t, p.t + \tau]$.

Since all blocking intervals have the same length τ, we only need to maintain the left endpoints of such intervals (using a balanced binary search tree) to find intersection counts. The number of blocking intervals is $O(n)$. Hence, insertion and query can both be finished in $O(\log n)$ time. The sorting takes $O(n \log n)$ time so the overall query time complexity of S-Base is $O(n \log n)$.

Next we describe two better algorithms that avoid sorting all records in the query interval.

B. Score-Band Algorithm (Monotone $\boldsymbol{f}$ Only)

If we could quickly find a small set of candidate records $\mathcal{C}$, which is guaranteed to be a superset of the answers; i.e.,

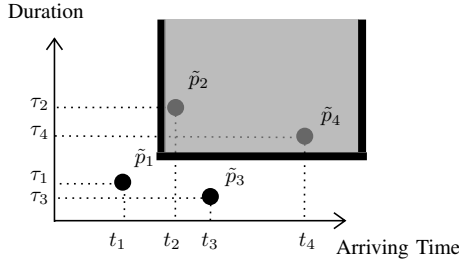**Fig. 4:** Index for k-skyband duration.

Algorithm 2: S-Band (k, I, τ)

Input: P, k, τ, and I.
Output: DurTop(k, I, τ)

1 $S \leftarrow \varnothing$, $\Gamma \leftarrow \varnothing$;
2 Compute $\mathcal{C} \subset P$ by finding durable k-skyband set;
3 Sort $\mathcal{C}$ in descending order of scores;
4 **for** $p \in \mathcal{C}$ **do**
5   **if** *p lies in* $< k$ *blocking intervals in* Γ **then**
6        $\pi_{\leq k} \leftarrow Q(k, [p.t - \tau, p.t])$;
7   **if** $p \in \pi_{\leq k}$ **then**
8           $S \leftarrow S \cup \{p\}$;
9   **else**
10   **for** $q \in \pi_{\leq k} \land q$ *not visited before* **do**
11             $\Gamma \leftarrow \Gamma \cup \{[q.t, q.t + \tau]\}$;
12    $\Gamma \leftarrow \Gamma \cup \{[p.t, p.t + \tau]\}$;
13 **return** S;

$S \subseteq \mathcal{C}$, then we could get a faster algorithm by only sorting $\mathcal{C}$. It is well-known that the k records with the highest score, with respect to *any monotone* scoring functions, belong to the k-skyband.[4] Hence, if a record p is τ-durable for a top-k query (with respect to a monotone f), then p must also be τ-durable for the k-skyband; i.e., p is in the k-skyband for the time interval $[p.t - \tau, p.t]$. This observation enables us to construct an offline index about each record's duration of belonging to the k-skyband, and efficiently produce a superset $\mathcal{C}$ of answers to durable top-k queries. Note that the score-band algorithm has its limitation, since the k-skyband technique only applies to monotone scoring functions.

Index. Score-Band algorithm needs additional index for finding candidate set $\mathcal{C}$, which we refer to as *durable k-skyband*. Suppose the value of k is known. For each record p, we compute the longest duration τ_p that p belongs to the k-skyband. Then we map each record p into the "arrival time - duration" plane as a two-dimensional point, $\tilde{p} = (p.t, \tau_p)$. We then index all such points in the 2D plane using a priority search tree [14] (or kd-tree, R-tree in practice). To answer DurTop(k, I, τ), we first ask a range query with the 3-sided rectangle $I \times [\tau, +\infty]$. The set of points that fall into the search region is the superset to actual answers of durable records. This index can be constructed in $O(n \log n)$ time, has $O(n)$

[4]For $\forall p, q, \in P$, p *dominates* q if p is no worse than q in all dimensions, and p is better than q in at least one dimension. k-skyband contains all the points that are dominated by no more than $k - 1$ other points. Skyline is a special case of k-skyband when $k = 1$.
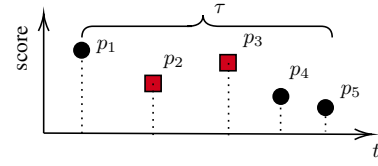

Fig. 5: Durability checks in S-Band and S-Hop.

space and the query time is $O(|\mathcal{C}| + \log n)$ in order to get the set $\mathcal{C}$. Figure 4 shows an example. We have four records p_1, p_2, p_3, p_4 arriving at t_1, t_2, t_3, t_4, whose duration for k-skyband is τ_1, τ_2, τ_3 and τ_4. We map them into $\tilde{p}_1, \tilde{p}_2, \tilde{p}_3$ and $\tilde{p}_4$ according to their arriving time and k-skyband duration. The 3-sided rectangle $I \times [\tau, +\infty]$ is shown as the shaded region. In this case, $\mathcal{C} = \{p_2, p_4\}$.

In general case, notice that we do not know the value of k upfront, i.e., a query has k as a parameter, so we cannot construct only one such index. There are two ways to handle it. If we have the guarantee that $k \leq \kappa_0$ for a small number κ_0 then we can construct κ_0 such indexes with total space $O(n\kappa_0)$. Otherwise, if k can be any integer in $[1, n]$, we can construct $O(\log n)$ such indexes (priority search trees), one for each $k = 2^0, 2^1, \ldots, 2^{\log n}$, so the space is $O(n \log n)$. Given a durable top-k query we first find the number $\bar{k}$ with $k \leq \bar{k} \leq 2k$, and then we use the corresponding index to get the superset $\mathcal{C}$. In this case, $\mathcal{C}$ contains the records that are τ-durable to the $\bar{k}$-skyband, so $S \subseteq \mathcal{C}$.

Query Algorithm. We refer to this score-prioritized approach using durable k-skyband candidates as Score-Band algorithm, or S-Band. Full algorithm is sketched in Algorithm 2 and described below. Given k, I, τ, we first retrieve the candidate set $\mathcal{C}$ using the durable k-skyband index as shown above. Then we sort $\mathcal{C}$ and visit records in descending order of their scores. For each record p we visit, we first check the number of blocking intervals that p lies. If p lies in less than k blocking intervals, it is a promising candidate and we run a top-k query on time interval $[p.t - \tau, p.t]$ for durability check. If p is indeed τ-durable, we add p to answer set. Otherwise, we need to add a blocking interval for each record returned by the top-k query (if we have not done so yet), since they all have higher scores than p. On the other hand, if p already lies in at least k blocking intervals, we can simply skip it. In the end, we add the blocking interval $[p.t, p.t + \tau]$ for p.

We can see that S-Band works similarly to S-Base. The only difference is that for a record that is blocked less than k times, we still have to execute a top-k query to check whether the record is τ-durable (Line 6). This step of durability check is necessary. Though some records are guaranteed to be non-durable (i.e., not captured by $\mathcal{C}$ with durable k-skyband), they can still block other records (with lower scores) to be durable ones. Consider a concrete example in Figure 5 where black dots represent candidate records in $\mathcal{C}$ and red squares represent records that are not in $\mathcal{C}$. S-Band would only visit p_1, p_4 and p_5. At the time we visit p_4, there is only one blocking interval (introduced by p_1). However, p_2 and p_3 actually have higher scores than p_4. By running a durability check query on p_4,

7

we can discover these missing records and add corresponding blocking intervals (Line 10-11) for better pruning power in future steps.

Complexity. The query time complexity of S-Band can be decomposed into three parts: 1) a range search query to find candidate set $\mathcal{C}$; 2) sort $\mathcal{C}$ according to their scores; 3) find durable records from sorted $\mathcal{C}$ sequentially. Summing up the above, the overall query time complexity of S-Band is $O\big(|\mathcal{C}|(q(n)+k)\log n\big)$, assuming that a top-$k$ query can be answered in $O(q(n)+k)$ time. In the worst case $|\mathcal{C}| = O(n)$ since all points can lie in the k-skyband. In Section V we show that using the probabilistic model in [15] (where the coordinates of the points are randomly assigned) the expected size of $\mathcal{C}$ is $O(k\lceil \frac{|I|}{\tau} \rceil \log^{d-1} \tau)$. Due to the blocking mechanism, in practice we expect that the number of top-k queries will be smaller. However, notice that we always need to sort all records in $\mathcal{C}$ which might make S-Band much slower due to the size of $\mathcal{C}$ that increases (in expectation) exponentially on the dimension d.

C. Score-Hop Algorithm

The data reduction strategy of S-Band offers adequate benefits for improving the overall running time on datasets in low dimensions (≤ 5). However, the query overhead on searching and sorting candidate records becomes a huge burden on high-dimensional data, as it is well-known that the size of k-skyband tends to explode (or equivalently, records in high-dimensional space tends to stay in k-skyband for a longer duration) in high-dimensional space. Furthermore, S-Band requires additional index and only applies to monotone scoring functions. To overcome the drawbacks of S-Base and S-Band, we propose another approach that does not require sorting and has better worst case guarantee. The main idea is that there is no need to sort records in advance; we can find the record with the next highest score one by one as we find durable records. With the help of blocking mechanism, we can skip certain time intervals when we find the next highest score record, despite the fact that there might be some high-score records in such intervals. This procedure has an analogy to the Time-Hop algorithm, since we effectively skip certain records while we traverse records in descending order of their scores, as we taking a hop in the score-domain.

Query Algorithm. We refer to this solution as Score-Hop algorithm, or S-Hop. The main idea of the algorithm is straightforward. In each iteration, we find the record with the maximum score among the records that lie in less than k blocking intervals. Let p be such a record. We run a durable top-k query so if p is a τ-durable record we add it in S. If p is not a τ-durable record, we add a blocking interval for each record returned by the durable top-k query (if they have not been added before). In the end, we add the blocking interval $[p.t, p.t + \tau]$ and we continue with the next record with the highest score. The actual implementation of the algorithm is more subtle, to guarantee a fast query time as described below; pseudo-code is provided in Algorithm 3. Given a query

---

Algorithm 3: S-Hop (k, I, τ)

Input: P, k, τ, and $I : [a, b]$.
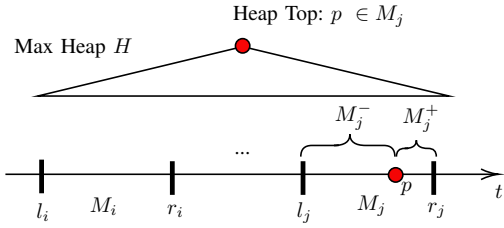**Output:** $\mathsf{DurTop}(k, I, \tau)$

1   $H \leftarrow \varnothing,\ S \leftarrow \varnothing,\ \Gamma \leftarrow \varnothing$;
2   **for** $[l_i, r_i]$: *disjoint τ-length intervals in I* **do**
3      $M_i \leftarrow Q(\mathbf{u}, k, [l_r, r_i])$;
4   H.push(M_i.pop());
5   **while** $H \neq \varnothing$ **do**
6      $p \leftarrow H$.pop(), and let $p \in M_j$;
7   **if** p *lies in* $< k$ *blocking intervals in* Γ **then**
8          $\pi_{\leq k} \leftarrow Q(\mathbf{u}, k, [p.t - \tau, p.t])$;
9   **if** $p \in \pi_{\leq k}$ **then**
10             $S \leftarrow S \cup \{p\}$;
11   **else**
12   **for** $q \in \pi_{\leq k} \wedge q$ *not visited before* **do**
13                 $\Gamma \leftarrow \Gamma \cup \{[q.t, q.t + \tau]\}$;
14          $M_j^- \leftarrow Q(k, [l_j, p.t - 1])$;
15          $M_j^+ \leftarrow Q(k, [p.t + 1, r_j])$;
16   H.push(M_j^-.top()), H.push(M_j^+.top());
17   **else if** $M_j \neq \varnothing$ **then**
18   H.push(M_j.pop());
19   **if** p *not visited before* **then**
20          $\Gamma \leftarrow \Gamma \cup \{[p.t, p.t + \tau]\}$;
21   **return** S;

---

interval $I = [a, b]$, we partition the interval into a set of disjoint τ-length sub-intervals: $[a, a + \tau), [a + \tau, a + 2\tau), \ldots, [a + \lfloor \frac{|I|}{\tau} \rfloor \tau, b]$. Let $[l_i, r_i]$ be the i-th sub-interval, and in each interval we find the k records [5] with the highest score, denoted M_i. We construct a max-heap H over all the top-1 records from all sub-intervals. Besides that, each node in H also keeps the original interval $[l_i, r_i]$ and the set M_i associated with the record. We repeat the following until H is empty. We take and pop the top record from H. Let p be that record originated from M_j. Then p will be processed in the following two cases: 1) If p lies in at least k blocking intervals, we update H by pushing the next top record in M_j (if there is any). 2) If p lies in less than k blocking intervals, we update H as follows. Assume that $[l_j, r_j]$ is the corresponding sub-interval of M_j (or p). We first split $[l_j, r_j]$ into two non-empty intervals $[l_j, p.t-1]$ and $[p.t+1, r_j]$. Then, run a top-k query on $[l_j, p.t-1]$ to get a new top-k set M_j^-. Similarly, get another new set M_j^+ from $[p.t+1, r_j]$. We replace the old set M_j with M_j^- and M_j^+, along with its corresponding interval $[l_j, p.t-1]$ and $[p.t+1, r_j]$, respectively. Finally, we update H by pushing the current top records from M_j^- and M_j^+ into the heap. In the end, we add the blocking interval from record p (if it is the first time we visited p). Figure 6 illustrates the main procedure of S-Hop on how to find next record with highest score. It is worth mentioning that the hopping movement happens at Line 18: we effectively skip certain intervals by not updating the max-heap and stop asking top-k queries on its sub-intervals.

---

[5]As a practical note, we notice that finding the top-1 record (instead of top-k) in each time interval can be more efficient in most real-life datasets.

Fig. 6: Illustration of Score-Hop algorithm on finding next record with highest score (if p lies in less than k blocking intervals).

[6]

Compared to S-Band, S-Hop does not have a strong dependency on the dimension of the data (only the running time of the top-k queries depends on the dimension) and makes better use of the blocking mechanism. In the end, we only find and process high-score records as we need instead of acquiring a full sorted order of records in advance, which leads to better worst case theoretical guarantees and faster query time. Experimental results in Section VI demonstrate that S-Hop can be 1 to 2 orders of magnitude faster than S-Band on high-dimensional (≥ 10) datasets.

Correctness.   The following lemma proves the correctness of S-Hop.

Lemma 2. *Given k, I and τ, the Score-Hop algorithm returns the correct answer for durable top-k query.*

Proof (Sketch). Let S^* be the τ-durable records in I. We show that $S \subseteq S^*$ and $S^* \subseteq S$. The algorithm always checks by running a top-k query if a record should be in the solution (line 8 of Algorithm 3) so $S \subseteq S^*$.

Next we prove $S^* \subseteq S$. The algorithm visits the records in descending (score) order so it is not possible that a record $p \in S^*$ lies in at least k blocking intervals before the algorithm visits p. We also need to prove that the algorithm does not miss any durable record in a sub-interval $[l_j, r_j]$ that corresponds to an empty M_j. If $|P([l_j, r_j])| \leq k$ then the result follows. Otherwise, we argue using induction that each time when the algorithm finds a record p in M_j that is contained in at least k blocking intervals, any timestamp in the sub-interval $[l_j, p.t]$ lies in at least k blocking intervals. Hence, if M_j is empty, any timestamp in $[l_j, r_j]$ lies in at least k blocking intervals and no other durable records are in $[l_j, r_j]$. ∎

D. Complexity Analysis of S-Hop

The query complexity analysis of S-Hop is non-trivial and needs more care. There are three main sub-procedures in S-Hop: find next highest score record, top-k queries for durability check and blocking mechanism. As presented above, the first two components both rely on multiple top-k queries. We first show a worst-case guarantee on the total number of top-k queries called in the algorithm. Please refer to Appendix C for full proof.

Lemma 3. *The total number of top-k queries performed by the Score-Hop algorithm is $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$.*

Proof (Sketch). As we had in the proof of Lemma 1 we need to bound the number of false checks. Let p be a false check and let p' be the record with the largest timestamp in $Q(k, [p.t - \tau, p.t])$. We say that p is assigned to p'. If $p'.t < a$, where a is the timestamp such that $I = [a, b]$, then we assign p to a. We first show that at the moment that we find the false check p the corresponding record p' can only have one of the following three properties: i) it lies in at least k blocking intervals, ii) $p' \in S$ and it lies in at most $k - 1$ blocking intervals, iii) $p' = a$. If p' has property ii) then p is a type-1 false check. Otherwise, p is a type-2 false check.

We first bound the number of type-1 false checks. Notice that after a type-1 false check p is assigned to p' then all timestamps in the sub-interval $[p'.t, p.t]$ lie in at least k records. So if another false check q later in the algorithm is assigned to p', again, then q can only be a type-2 false check. Hence, the type-1 false checks are bounded by $O(|S|)$. In order to bound the type-2 false checks we assume a window ρ of length τ in I. We make the following key observation: At the moment that we find a type-2 false check p, it lies in at most $k - 1$ blocking intervals while p' lies in at least k blocking intervals, so there should be a blocking interval $[l, r]$, where its right endpoint lies between $p'.t$ and $p.t$, i.e., $p'.t \leq r \leq p.t$. (Notice that if $p' = a$ is assigned more than once then it also lies in at least k blocking intervals.) Using this observation along with other properties of the false checks we can show that after finding k type-2 false checks in ρ, each timestamp in ρ will lie in at least k blocking intervals. Hence, the algorithm will not run any other top-k query in ρ. Since there are $\lceil\frac{|I|}{\tau}\rceil$ disjoint τ-length sub-intervals in I we can bound the total number of type-2 false check by $O(k\lceil\frac{|I|}{\tau}\rceil)$. Overall, the number of false checks along with the durable records in I is $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$. ∎

The lemma above also shows that the number of different sets M_j that are created by the algorithm is $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$. For each set we can visit at most k records so in total the algorithm may visit $O(k(|S| + k\lceil\frac{|I|}{\tau}\rceil))$ records [7]. Each top(), or pop() procedure takes $O(\log n)$ time so in total we need $O(k(|S| + k\lceil\frac{|I|}{\tau}\rceil)\log n)$ to visit these records. Furthermore, recall that we need $O(\log n)$ time to check if a record lies in at least k blocking intervals and $O(\log n)$ time to insert a blocking interval (using a binary search tree) so we also spend $O(k(|S| + k\lceil\frac{|I|}{\tau}\rceil)\log n)$ time for the blocking mechanism. Notice that this running time is dominated by the time to answer $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$ top-k queries, so S-Hop answers a durable preference top-k query in $O\big((|S| + k\lceil\frac{|I|}{\tau}\rceil)(q(n) + k)\log n\big)$ time (with an efficient top-k query procedure in $O(q(n) + k)$).

---

[6]In practice, we make sure that when we ask k top-1 queries in an interval we remove it from the max-heap.

[7]We note that the algorithm may visit some records, that lie in at least k blocking intervals, more than once. The upper bound $O(k(|S| + k\lceil\frac{|I|}{\tau}\rceil))$ counts all the times that the algorithm visits a record. We can modify the algorithm so that it does not visit the same record twice but that would make the description of the algorithm more complicated without decreasing the overall asymptotic complexity.

Similarly to T-Hop our index for S-Hop has near linear space and supports updates in polylogarithmic time for preference queries.

As it turns out, hopping in time-domain (T-Hop) and in score-domain (S-Hop) gives us the same complexity bound. But in practice, S-Hop is more conservative in asking preference top-k queries compared to T-Hop, due to the candidate pruning brought by blocking mechanism. This makes S-Hop run faster than T-Hop when the top-k query itself is expensive; i.e., a larger k or on high-dimensional datasets.

V. EXPECTED COMPLEXITY

In the previous sections we presented two types of algorithms (time-prioritized and score-prioritized) to answer durable top-k queries with the same worst-case guarantee on their query time. In particular we showed that their query times depend on $k\lceil\frac{|I|}{\tau}\rceil$ and $|S|$. In this section, we go beyond the worst-case analysis and analyze their performance in a more "expected" sense. Most importantly, we show in Section V-A that the expected size of $|S|$ is roughly $k\lceil\frac{|I|}{\tau}\rceil$ if the scores of data records are drawn randomly from an *arbitrary* distribution (which can be picked by a powerful adversary with the advance knowledge of the query parameters). This result essentially establishes that, under this model, our best algorithms are in a sense optimal because their complexity is expected to be linear in the output size. Secondly, in Section V-B, we study the expected complexity of Score-Band algorithm by bounding the expected size of τ-durable k-skyband candidate set $\mathcal{C}$ using the same probabilistic model used in [15].

A. Expected Answer Size

Consider a set of n records P with $p_i.t = i$, for $p_i \in P$. We analyze the expected size of a query output when the scores of records are assigned in a semi-random manner, where the data values can be arbitrarily chosen and then they are assigned in a random order to the records. More formally, we consider a *random permutation model* (RPM). Let $\mathbf{X} = x_1 < x_2 < \ldots < x_n$ be a sequence of n arbitrary non-negative numbers chosen by an adversary, and let σ be a permutation of $\{1, \ldots, n\}$. We set $f(p_i) = x_{\sigma(i)}$, i.e., the score of record p_i is $x_{\sigma(i)}$, where $\sigma(i)$ is the image of i under σ. As argued in [16], the random permutation model is more general than the model in which all scores are drawn from an arbitrary unknown distribution, so our result holds for this model as well. The random permutation model has been widely used in a rich variety of domains and considered as a standard for complexity analysis; i.e., online algorithms [17]–[19], discrete geometry [20]–[22], and query processing [16]. Our main result is the following.

Lemma 4. *In the random permutation model, given k, τ and I, we have $\mathbf{E}[|S|] = k\frac{|I|}{\tau+1}$.*

Proof. For a record $p_i \in P(I)$, let X_i be the random variable, which is 1 if p_i is a τ-durable record, and 0 otherwise.

Thus, $\mathbf{E}[|S|] = \mathbf{E}[\sum_i X_i]$. Using the linearity of expectation, $\mathbf{E}[\sum_i X_i] = \sum_i \mathbf{E}[X_i] = \sum_i \mathbf{Pr}[X_i = 1]$.

Thus our goal is to compute $\mathbf{Pr}[X_i = 1]$: the probability that there are less than k records in $[p_i.t - \tau, p_i.t)$ with score larger than $f(p_i)$. Let $P_i^\tau = \{p_{i-\tau}, \ldots, p_{i-1}\}$. For a subset $Q \subset P_i^\tau$, let A_Q be the binary random variable, which is 1 if all records in Q have score greater than $f(p_i)$ and all records in $\overline{Q} = P_i^\tau \setminus Q$ have score less than $f(p_i)$. We have

$$\mathbf{Pr}[X_i = 1] = \sum_{l=0}^{k-1} \sum_{Q \subset P_i^\tau, |Q|=l} \mathbf{Pr}[A_Q]. \quad (1)$$

We estimate $\mathbf{Pr}[A_Q]$ as follows. Let $V \subset \mathbf{X}$ with $|V| = \tau + 1$. We first bound the conditional probability $\mathbf{Pr}[A_Q \mid V]$ such that the records in $P_i^\tau \cup \{p_i\}$ are assigned scores from V. We consider all possible permutations of V and count only those cases where the records in Q have larger value than $f(p_i)$, and the records in $\overline{Q}$ have values less than the value of $f(p_i)$. Notice that the permutations that satisfy this property must assign the first l largest values of V to Q, then the $(l+1)$-th largest value to p_i and the rest $\tau - l$ smaller values of V to $\overline{Q}$. Under such assignment, any permutations of values in Q and $\overline{Q}$ are valid cases. Hence, the number of valid permutations are $l!(\tau - l)!$, while the number of all possible permutations of V are $(\tau + 1)!$. We have

$$\mathbf{Pr}[A_Q \mid V] = \frac{l!(\tau - l)!}{(\tau + 1)!} = \frac{1}{\tau + 1}\frac{1}{\binom{\tau}{l}}. \quad (2)$$

Since (2) holds for all V, $\mathbf{Pr}[A_Q] = \frac{1}{\tau + 1}\frac{1}{\binom{\tau}{l}}$. Substituting this in (1), we obtain

$$\mathbf{Pr}[X_i = 1] = \sum_{l=0}^{k-1}\binom{\tau}{l}\frac{1}{\tau+1}\frac{1}{\binom{\tau}{l}} = \sum_{l=0}^{k-1}\frac{1}{\tau+1} = \frac{k}{\tau+1} \quad (3)$$

Finally,

$$\mathbf{E}[|S|] = \sum_i \mathbf{Pr}[X_i = 1] = k\frac{|I|}{\tau + 1}. \quad (4)$$

$\square$

Combining Lemma 4 with the analysis of Sections III-C and IV-D, we conclude that in a random permutation model the *expected* query time complexity of both Time-Hop and Score-Hop algorithms is $O(|S|(q(n) + k)\log n)$, or equivalently $O\big(k\lceil\frac{|I|}{\tau}\rceil(q(n) + k)\log n\big)$, where $O(q(n) + k)$ reflects the time complexity of answering a top-k query. In Section VI, our experimental results on real and synthetic datasets both confirm this finding.

B. Expected size of durable k-skyband

In this subsection we bound the expected size of τ-durable k-skyband records, denoted by $\mathcal{C}$, from Section IV-B in a probabilistic model similar to the previous case. Recall that the size of $\mathcal{C}$ affects the running time of the S-Band algorithm.

Let $P = \{p_1, \ldots, p_n\}$ with $p_i.t = i$. We use the same random model as in [15] where (the attributes of) records

TABLE II: Dataset summary

Dataset	Dimensionality	Size (# records)
NBA-X	1,2,3,5	1M
Network-X	2,3,5,10,20,30,37	5M
Syn-X	2	1M,2M,5M,10M,20M,50M

are randomly generated. The following lemma bounds the expected size of $\mathcal{C}$ (See Appendix D).

Lemma 5. *In the random model as in [15], given k, τ and I, we have* $\mathbf{E}\left[|\mathcal{C}|\right] = O(k\frac{|I|}{\tau}\log^{d-1}\tau)$.

Combining Lemma 5, the analysis of Section IV-B and an efficient top-k query procedure runs in $O(q(n) + k)$ time, the *expected* query time complexity of Score-Band algorithm is $O\left(k\lceil\frac{|I|}{\tau}\rceil(q(n)+k)\log n \log^{d-1}\tau\right)$. It shows that the expected complexity of Score-Band algorithm can be higher than Time-Hop or Score-Hop algorithm by a factor of at most $\log^{d-1}\tau$. Experimental results in Section VI also confirm this finding as we vary the data dimensionalities. The curse of dimensionality makes Score-Band algorithm perform worse even compared to other simple baselines. Again, Time-Hop and Score-Hop are both generally applicable to arbitrary user-specified scoring functions, while Score-Band only works for monotone functions.
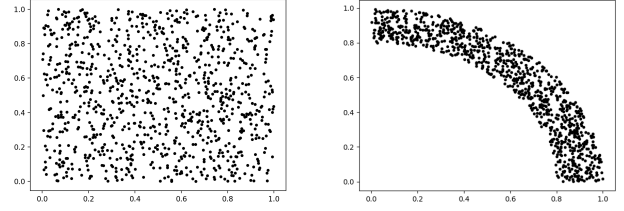
VI. EXPERIMENTS

A. Experiment Setup

Datasets. We use two real-life datasets and some synthetic ones, as summarized in Table II and described below:

NBA[8] contains the performance of *each* NBA player in *each* game from 1983 to 2019, with in total ~ 1 million individual performance records on 15 numeric attributes. Records are naturally organized by date and time, and we break ties (e.g., performances of different players in the same game) arbitrarily. We choose some subsets of 15 attributes to create datasets with different dimensions collectively referred to as **NBA-X**: **NBA-1** selects only *3-point-made*; **NBA-2** captures the *points* and *assists*; **NBA-3** chooses *points*, *assists*, *rebounds*; **NBA-5** includes five dimensions: *points*, *assists*, *rebounds*, *steals* and *blocks*.

Network[9] is the dataset from KDD Cup 1999. This dataset contains ~ 5 million records with 37 numeric attributes that describe network connections to a machine, including *connection duration*, *packet size*, etc. The query in this case utilizes a scoring function that weighs a variety of numerical attributes to rank connections in order to identify unusual and potentially malicious ones. Records have unique timestamps and are ordered by these timestamps. Since these attributes have different measurement units, we scale the value of each dimension using MinMax normalization. To study the impact of data dimensionalities on query efficiency, we choose the first 2, 3, 5, 10, 20, 30 and 37 attributes from the full dimensions to create 7 different datasets collectively referred

[8]NBA datasets were collected from https://www.basketball-reference.com/
[9]https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html

| **(1)** IND | **(2)** ANTI |

Fig. 7: Value distributions for synthetic dataset

TABLE III: Query Parameters (default value in bold)

Parameter	Range
k	5, **10**, 15, 20, 25, 30, 35, 40, 45, 50
τ	1%, 5%, 10%, 15%, **20%**, 25%, 30%, 40%, 50%
$|I|$	10%, 20%, 30%, 40%, **50%**, 60%, 70% 80%
d	1, **2**, 3, 5, 10, 20, 30, 37

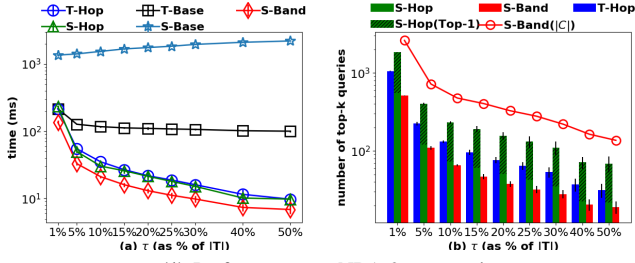to as **Network-X**, where **X** represents the dimensionality of the dataset.

Syn is a synthetic two-dimensional dataset that is used for scalability test on proposed solutions. We generate Syn with independent (IND) and anti-correlated (ANTI) data distributed in a 2D unit square. For IND data, the attribute values of each tuple are generated independently, following a uniform distribution. ANTI data are drawn from the portion inside the positive orthant of an annulus centered at the origin with outer radius 1 and inner radius 0.8, representing an environment that most of the records gather in k-skyband. Figure 7 illustrates the sample value distributions of IND and ANTI. The full size of Syn is 50 million and each data point has an unique arriving time. We further choose several subsets of Syn with 1, 2, 5, 10 and 20 millions of records. The set of synthetic datasets are collectively referred to as **Syn-X**, where **X** represents data size.

Query Parameters. Table III summaries the query parameters under investigation, along with their ranges and default values. Among these, the query interval length $|I|$ and the durability τ is measured as percentage of dataset size n. When varying query interval length, we always fix the right endpoint of the interval to be the most recent timestamp in dataset and only move the left endpoint.
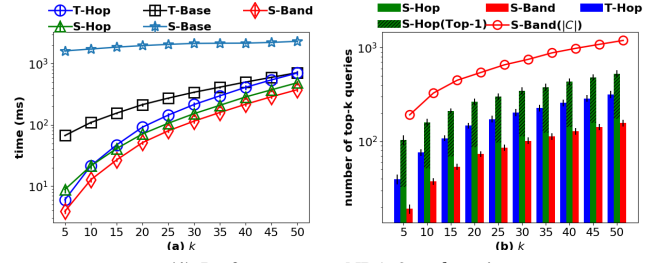
Implementations & Evaluation Metric. To make the discussions concrete and concise, we choose a linear and monotone preference scoring function throughout the experimental section in the simple form: $f_{\mathbf{u}}(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$, where $\mathbf{u}$ is a user-specified preference vector and $\mathbf{u}_i$ is the (non-negative) weight for i-th attribute of a record. At query time, user need to specify $\mathbf{u}$ as one of the input parameters. Since the focus of this paper is not to develop the best possible index for top-k queries $Q_{\mathbf{u}}(k, W)$, our implementation of the top-k building block simply adopts a tree index (on the time domain of P), and answer $Q_{\mathbf{u}}(k, W)$ in a straightforward top-down manner with a branch-and-bound method. More specifically, each tree node stores the *skyline* of all records that it contains. The skyline helps us quickly identify the maximum score of each node under *any* preference vector $\mathbf{u}$. Then, to answer $Q_{\mathbf{u}}(k, W)$,
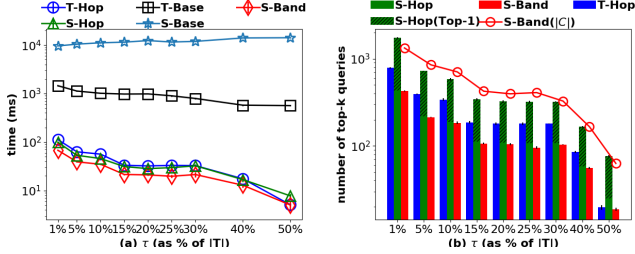
(1) Performance on NBA-2 as τ varies.
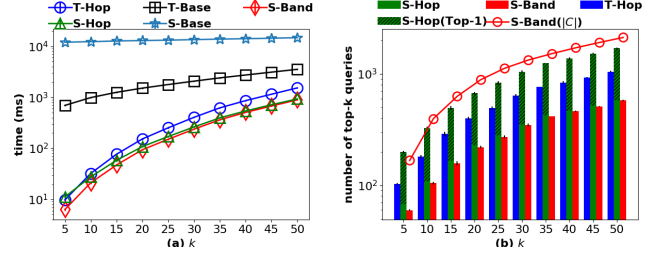

(1) Performance on NBA-2 as k varies.

(2) Performance on Network-2 as τ varies.

Fig. 8: Performance comparison as τ varies.

(2) Performance on Network-2 as k varies.

Fig. 9: Performance comparison as k varies.

it is sufficient to use at most k nodes (that are contained by time window W) with the highest scores according to **u**. This index offers adequate performance in our experiments, but it can certainly be replaced by more sophisticated index with better worst-case guarantees, without affecting the rest of our proposed solution.

Using the building block of top-k queries described above, we further implement **T-Base** (Section III-A), **T-Hop** (Section III-B), **S-Base** (Section IV-A), **S-Band** (Section IV-B) and **S-Hop** (Section IV-C). Performance of various methods are evaluated using the following two metrics: number of top-k queries and overall query time (in millisecond). For each query parameter setting, we run the query 100 times with 100 different randomly generated preference vectors, and report the average with standard deviation.

All methods were implemented in C++, and all experiments were performed on a Linux machine with two Intel Xeon E5-2640 v4 2.4GHz processor with 256GB of memory.
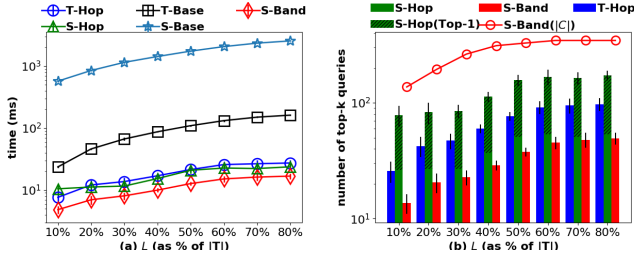
B. Algorithm Evaluations

According to the theoretical analysis of our algorithms in previous sections, the query efficiency depends on the length of durability window τ, the value of k, the length of query interval I, the data dimensionality d and the data size n. For fair evaluation and comparison of algorithm efficiency, we designed a set of variable-controlling experiments such that each time we only vary one query parameter of interest and fix the others to default values.
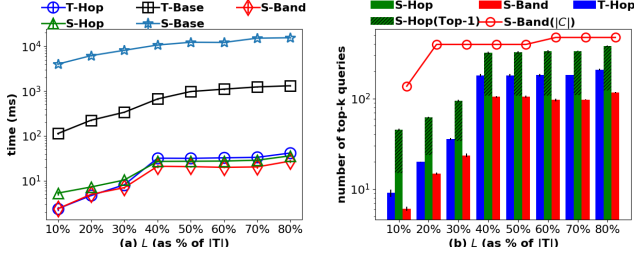
Comparison of Algorithms when Varying τ. In Figure 8, we investigate the performance of all durable top-k solutions, as we vary durability τ. Figure 8-1-(a) shows the query efficiency comparison on NBA-2. The sorting based solution S-Base is the slowest, as it requires fully sorting all records in the time interval of length $|I| + \tau$. T-Base is faster than S-Base and mostly independent of τ. All the rest solutions, T-

Hop, S-Hop and S-Band, become more efficient as we increase τ, or equivalently, when query is more selective. This finding confirms our analysis in Section V that the query efficiency bounds of Hop-based solutions and S-Band both depend on the answer size, which is $O(k\frac{|I|}{\tau})$. T-Hop and S-Hop nearly perform the same, while S-Band can be slightly faster. When the query is highly selective (τ is half of the length of entire time domain), they are 1-2 orders of magnitude faster compared to T-Base and S-Base, respectively. Similar trends can be seen in Figure 8-2-(a), where we test algorithms on a larger dataset Network-2. The only difference is that baseline solutions (T-Base and S-Base) are more expensive and the efficiency difference between baseline solutions and T-Hop/S-Hop/S-Band is even larger (up to 3 orders of magnitude).

Next, we take a closer look at T-Hop, S-Hop and S-Band in Figure 8-1-(b), which compares the number of top-k queries needed for these three advanced algorithms. For S-Hop, the total number of top-k queries is decomposed into two parts: top-k queries for durability check (unshaded region of a green bar) and top-k queries for finding the next highest score record (shaded region). For S-Band, we also plot the size of durable k-skyband candidate set C on top the figure as red circled line, reflecting the overhead cost of sorting C for S-Band. Now it is clear that the main reason why T-Hop/S-Hop/S-Band becomes faster when τ is large is that fewer top-k queries are needed. A more selective query with larger τ also makes the candidate set C of S-Band smaller, demonstrating the effectiveness of using durable k-skyband to identify promising candidates. On the other hand, we can see that S-Hop and S-Band ask fewer top-k queries than T-Hop, demonstrating the pruning power of blocking mechanism in score-prioritized solutions. This figure also explains why S-Band runs slightly faster than S-Hop and T-Hop on NBA-2 in this case, as S-Band requires the least number of top-k queries and the overhead cost on sorting candidate set C is relatively small on two-dimensional data.

(1) Performance on NBA-2 as $|I|$ varies.

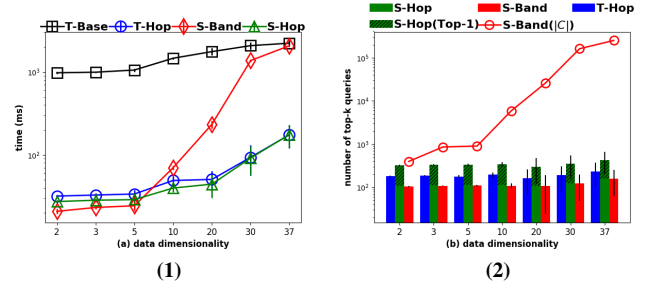**(2)** Performance on Network-2 as $|I|$ varies.

Fig. 10: Performance comparison as $|I|$ varies.

(1)

(2)

Fig. 11: Performance comparison on Network-X as d varies.

(1) IND

(2) ANTI

Fig. 12: Scalability test on IND and ANTI Syn-X.

Again, similar trends can be found in Figure 8-2-(b).

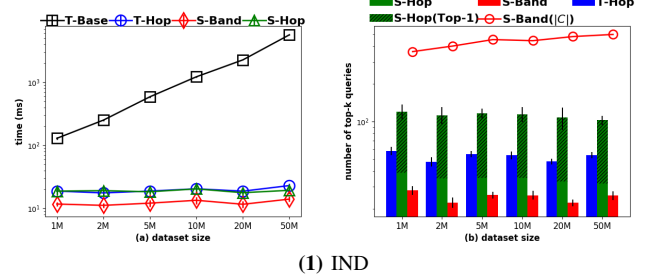**Comparison of Algorithms when Varying k.** Next, we study the effect of k on efficiency. Results are shown in Figure 9. When we increase k, not only need we ask more top-k queries (see Figure 9-1-(b) and Figure 9-2-(b)), but a top-k query itself also becomes more expensive. Thus in both Figure 9-1-(a) and Figure 9-2-(a), we can see that all algorithms (except S-Base) are slower when k is larger. Especially when k reaches 50, top-k computations become the dominant factor on overall efficiency, and the differences among the various algorithms diminish. Still, S-Band and S-Hop have slight advantages over T-Hop on larger k, as they use blocking mechanism to prune candidate records and are more conservative in asking expensive top-k queries.

Comparison of Algorithms when Varying $|I|$. In Figure 10, we compare the performance of proposed algorithms as we vary the query interval length $|I|$. In terms of efficiency, Figure 10-1-(a) and Figure 10-2-(a) show that T-Hop/S-Hop/S-Band is much faster than baseline solutions T-Base and S-Base, especially on the large dataset Network-2. On the other hand, we also find that our proposed algorithms scale better with $|I|$ than with k (recall Figure 9). The reason is that the time complexities of T-Hop/S-Hop and S-Band are quadratic in k but only linear on $|I|$ (recall Lemma 4 and Lemma 5). In terms of number of top-k queries, in Figure 10-1-(b) and Figure 10-2-(b), it is not surprising to see that all proposed solutions ask more top-k queries as $|I|$ increases. The relative performance of various algorithms is consistent with previous experiments where we varied τ or k.
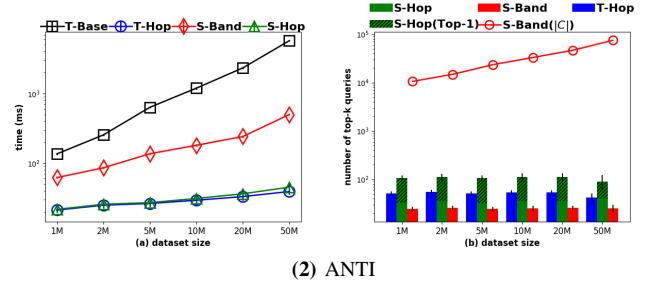
Comparison of Algorithms when Varying d. In this section, we study the effect of data dimensionality d on algorithm performances. Since the sorting-based S-Base is clearly inferior to other algorithms, here we only test T-Base, T-Hop, S-Band and S-Hop on Network-X with varying dimensions. Results are shown in Figure 11. Let us first take

a look on Figure 11-2. We can see that the number of top-k queries for all proposed algorithms stays stable as we increase dimensionality. This finding again confirms our theoretical analysis that the number of top-k queries (or, answer size) depends only on $k\frac{|I|}{\tau}$ and is independent of dimensionality d. On the other hand, we can see that the size of candidate set C for S-Band rockets in high dimensions, and can be up to 4 orders of magnitude larger than the size of actual promising records. The sorting overhead on such huge candidate sets is already too big. Then, let us go back to Figure 11-1. The query time of T-Base, T-Hop and S-Hop slowly increases as we increase dimensionality, because top-k queries on high-dimensions become more expensive, yet they ask roughly the same number of top-k queries regardless of dimensionality. While S-Band still performs well on low-dimensional data (less than 5 dimensions), in higher dimension S-Band becomes dramatically worse, even taking as much time as T-Base on Network-37.

Scalability. Finally, we use the two-dimensional synthetic dataset Syn-X to test the scalability of the proposed algorithms as we vary the input size from 1 million to 50 million. Figure 12 summarizes the results. As the input size increases, we also increase the query interval length proportionally (so it remains at a fixed percentage of the data size). As shown in Figure 12-1, we can see that T-Hop, S-Hop and S-Band scale well on large IND datasets, and S-Band again performs slightly
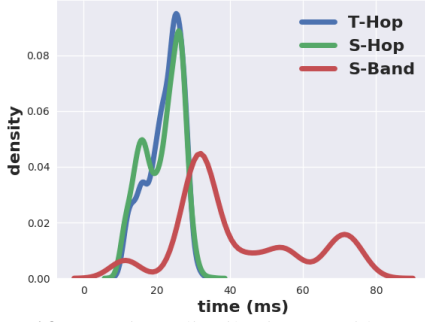
Fig. 13: Runtime distribution on 5d NBA data.

TABLE IV: Query time (in seconds) comparison on NBA-2 when varying τ. PostgreSQL backend.

τ (as % of $|T|$)	10%	20%	30%	40%	50%
T-Hop	0.46	0.28	0.18	0.12	0.1
T-Base	2.2	1.9	1.8	1.7	1.7

TABLE V: Query time (in seconds) comparison on NBA-2 when varying $|I|$. PostgreSQL backend.

$|I|$ (as % of $|T|$)	10%	20%	30%	40%	50%
T-Hop	0.1	0.16	0.17	0.2	0.26
T-Base	0.46	0.93	1.3	1.6	2

better than T-Hop and S-Hop. The running time of S-Base increases on larger datasets simply because we are also making the query interval longer. Figure 12-1-(b) further illustrates that the total number of top-k queries asked by different algorithms is also independent from the data size. A larger dataset only makes top-k queries more expensive. Although the size of candidate set $|C|$ increases on larger IND datasets, its growth rate here is much lower than its growth rate when varying dimensionality d in Figure 11. Overall, on IND synthetic data, $|C|$ is only about 4-5 times bigger than the actual answer size, which will not incur a big sorting overhead for S-Band. However, the situation is much different for ANTI Syn-X. As shown in Figure 12-2, in terms of query efficiency, T-Hop and S-Hop still scale well, but S-Band now becomes much more expensive because of the data distribution of ANTI. Most records in ANTI data would gather in k-skyband, resulting in C up to 3 orders of magnitude larger than the actual answer size (see Figure 12-2-(b)), which hurts the performance of S-Band. The efficiency of S-Band has a strong dependency on the candidate set C, or more generally, the data distribution. In contrast, the performance of T-Hop and S-Hop in this case is nearly independent of both size and distribution of data; it is only linear to the answer size.

Query Time Distribution over Different Real Datasets. Figure 12 already clearly illustrates the performance difference of S-Band on IND and ANTI synthetic data, demonstrating the effect of data distributions on S-Band's query efficiency. Here, we further compare T-Hop, S-Hop and S-Band on real data, and study how data distributions would influence their performance in practice. We use NBA as the main data source, and select 20 combinations of 5 dimensions randomly chosen out of the 15 attributes, e.g., (points, assits, rebounds, steals, blocks), (points, assits, steals, blocks, 3-pointers-made), etc. These resulting 20 datasets have the same dimensionality (5) but exhibit different distributions. We run queries with default settings on each dataset, and plot the running time distribution for all datasets. Results are shown in Figure 13. We can see that S-Band takes longer time on average, and also has a wide span on query time. This finding again confirms that S-Band is highly sensitive to underlying data distributions. In contrast, running times of T-Hop and S-Hop are centered in narrower value ranges, showing their robustness to data distributions and further demonstrating their advantages over S-Band on

real data.

In sum, we conclude that the Hop-based algorithms, T-Hop and S-Hop, are the best solutions for answering durable preference top-k queries. They scale well on large datasets as well as to high dimensions, and most importantly, their query time complexity is proportional to the answer size. This property makes T-Hop and S-Hop run even faster when the query is highly selective; i.e., smaller k or larger τ, which tend to be the more practical and meaningful query settings that people would use in real-life applications. While S-Band is also a reasonable approach, its performance depends highly on the data characteristics (faring poorly in high dimensions and for certain distributions). S-Band also requires additional offline indexing for finding durable k-skyband candidates. Overall, as demonstrated by experiments on both real and synthetic data, efficiency and robustness of Hop-based solutions make them more attractive solutions. Even on very large and high-dimensional datasets, T-Hop/S-Hop only need less than a second to return durable top records for any given preference, which enables interactive data exploration.

C. DBMS-Based Implementations

To demonstrate the generality of proposed solutions and its possibility of integrating into a DBMS, we further test the algorithms utilizing PostgreSQL [23] as the backend DBMS. More specially, we load the datasets NBA-2, Syn-500M (IND) and Syn-500M (ANTI) into PostgreSQL tables. The table schema consist of numeric attributes of the records and an additional column representing arriving time instant. For algorithm implementations, we code T-Hop and T-Base as stored procedures using PL/Python with PostgreSQL's native support operators.[10] Besides data tables, we also create corresponding index tables to support efficient top-k records retrieval. The index table is similar to the tree-based index as we used for previous experiments, providing sufficient data reduction for answering range top-k queries. Again, the top-k module

---

[10] The other proposed solution, S-Hop, requires a more delicate query procedure and data structures (recall Algorithm 3). Hence it is more suitable to implement S-Hop as a wrapper function outside the DBMS.

TABLE VI: Query time (in seconds) comparison on different datasets. Dataset size (measured by DBMS storage size) is shown in parentheses. PostgreSQL backend.

Dataset	NBA-2 (**0.05 G**)	Syn-IND (**30 G**)	Syn-ANTI (**30 G**)
T-Hop	0.28	1.9	2.3
T-Base	1.9	773	787

can be replaced by more sophisticated indexes with better performance, without affecting the rest of our solution.

Tables IV and V show the results of testing T-Hop and T-Base on the smaller NBA-2 dataset with the same query setting as before, varying durability τ and query interval length $|I|$ to compare query efficiencies. Similar conclusions can be drawn here. T-Base always pays linear cost (continuous sliding windows) to visit all records in the query interval. Thus, the running time is linear to $|I|$ (Table V), and nearly independent of τ (Table IV). In comparison, T-Hop's complexity is linear to the answer size, which makes it run faster as query becomes more selective (smaller $|I|$ or larger τ). Overall, T-Hop is at least $10\times$ faster than T-Base.

In Table VI, we increase the dataset size up to 500M records, which takes around 30 Gigabytes of disk space in PostgreSQL. Running default queries in such cases, we can see that T-Hop is more than $100\times$ faster than T-Base, bringing down the query time from nearly 12 minutes to just 2 seconds. T-Hop also apparently scales well on large datasets, since the complexity is mostly linear to the answer size. The query time increase solely comes from the more expensive top-k module. On the contrary, the continuous sliding-window nature of T-Base makes it prohibitively slow when dealing with large amounts of temporal data.

D. Summary of Experiments

In sum, we conclude that the Hop-based algorithms, T-Hop and S-Hop, are the best solutions for answering durable preference top-k queries. They scale well on large datasets as well as to high dimensions, and most importantly, their query time complexity is proportional to the answer size. This property makes T-Hop and S-Hop run even faster when the query is highly selective; i.e., smaller k or larger τ, which tend to be the more practical and meaningful query settings that people would use in real-life applications. While S-Band is also a reasonable approach, its performance depends highly on the data characteristics (faring poorly in high dimensions and for certain distributions). S-Band also requires additional offline indexing for finding durable k-skyband candidates. Overall, as demonstrated by experiments on both real and synthetic data, efficiency and robustness of Hop-based solutions make them more attractive solutions. Even on very large and high-dimensional datasets, T-Hop/S-Hop only need less than a second to return durable top records for any given preference, which enables interactive data exploration. Finally, T-Hop can be efficiently implemented inside a DBMS; for large datasets (tens of Gigabytes), it brings down the query time to just a couple of seconds, from more than 10 minutes required without our solution.

VII. Related Work

The notion of "durability" on temporal data has been studied by previous works, but they consider different definitions of durability and/or different data models from ours. In [24] and [25], authors implicitly considered "durability" in the form of prominent streaks in sequence data, and devised efficient algorithms for discovering such streaks. Given a sequence of values, a prominent streak is a long consecutive subsequence consisting of only large (small) values. Their algorithms can also be extended to find general top-k, multi-sequence and multi-dimensional prominent streaks. Jiang and Pei [26] studied Interval Skyline Queries on time series, which can be viewed as another type of "durability" when segments of time series dominate others.

Another line of durability-related work on temporal data is represented by [27]–[29] and [30]. Consider a time-series dataset with a set of objects, where the data values of each object are measured at regular time intervals; i.e., stock markets. At each time t, objects are ranked according to their values at t. The definition of "durability" therein is the fraction of time during a given time window when an object ranks k or above. This line of work mainly focused on how to efficiently aggregate rankings (rank $\leq k$ or not) over time. [30] applied durable top-k searches in document archives, finding documents that are consistently among the most relevant to query keywords throughout a given time interval. In that setting, the challenge is how to merge multiple per-keyword relevance scores over time efficiently into a single rank.

Durable queries also arise in dynamic or temporal graphs, typically represented as sequences of graph snapshots. For example, in [31] and [32], authors considered the problem of finding the (top-k) most durable matches of an input graph pattern query; that is, the matches that exist for the longest period of time. The main focus is on the representations and indexes of the sequence of graph snapshots, and how to adapt classic graph algorithms in this setting.

Besides durability, Mouratidis et al. [11] studied how to continuously monitor top-k results over the most recent data in a streaming setting. Our baseline solution used in Section VI shares the same spirit as algorithms in [11] for incrementally maintaining top-k results over consecutive sliding windows.

VIII. Conclusion

In this paper, we have initiated a comprehensive study into the problem of finding durable top records in large instant-stamped temporal datasets by running durable top-k queries. We proposed two types of novel algorithms for efficiently solving this problem, and provided in-depth theoretical analysis on the complexity of the problem itself and of our algorithms. As demonstrated by experiments on real and synthetic data, our best solutions, Time-Hop and Score-Hop, find interesting durable top records in under a second on large and high-dimensional datasets, and can be up to 2 orders of magnitude faster than existing baselines.

References

[1] P. Afshani and T. M. Chan, "Optimal halfspace range reporting in three dimensions," in *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, 2009.

[2] B. Chazelle, L. J. Guibas, and D.-T. Lee, "The power of geometric duality," *BIT Numerical Mathematics*, vol. 25, 1985.

[3] J. Matousek, "Reporting points in halfspaces," *Computational Geometry*, vol. 2, 1992.

[4] P. K. Agarwal *et al.*, "Efficient searching with linear constraints," *J. Comp. and System Sciences*, vol. 61, 2000.

[5] P. K. Agarwal and J. Matoušek, "Dynamic half-space range reporting and its applications," *Algorithmica*, vol. 13, 1995.

[6] T. M. Chan, "Three problems about dynamic convex hulls," *International Journal of Computational Geometry & Applications*, vol. 22, 2012.

[7] Y.-C. Chang *et al.*, "The onion technique: indexing for linear optimization queries," in *SIGMOD*, vol. 29, 2000.

[8] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen, "Efficient maintenance of materialized top-k views," in *ICDE*, 2003.

[9] V. Hristidis and Y. Papakonstantinou, "Algorithms and applications for answering ranked queries using ranked views," *VLDB J.*, vol. 13, 2004.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *CSUR*, 2008.

[11] K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of top-k queries over sliding windows," in *SIGMOD*, 2006.

[12] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin, "Sliding-window top-k queries on uncertain streams," *VLDB*, vol. 1, 2008.

[13] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas, "Ad-hoc top-k query answering for data streams," in *VLDB*, 2007.

[14] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, "Computational geometry," in *Computational geometry*, 1997.

[15] J. L. Bentley, H.-T. Kung, M. Schkolnick, and C. D. Thompson, "On the average number of maxima in a set of vectors and applications." CARNEGIE-MELLON UNIV, Tech. Rep., 1977.

[16] P. K. Agarwal *et al.*, "Range-max queries on uncertain data," *Journal of Computer and System Sciences*, vol. 94, 2018.

[17] G. Goel and A. Mehta, "Online budgeted matching in random input models with applications to adwords," in *Proc. 19th Annual ACM-SIAM Symp. on Discrete algorithms*, 2008.

[18] M. Mahdian and Q. Yan, "Online bipartite matching with random arrivals: an approach based on strongly factor-revealing lps," in *Proc. 43rd Annual ACM Symp. on Theory of computing*, 2011.

[19] A. Mehta, A. Saberi, U. Vazirani, and V. Vazirani, "Adwords and generalized on-line matching," in *FOCS*, 2005.

[20] P. K. Agarwal, H. Kaplan, and M. Sharir, "Union of hypercubes and 3d minkowski sums with random sizes," in *ICALP*, 2018.

[21] P. K. Agarwal, S. Har-Peled, H. Kaplan, and M. Sharir, "Union of random minkowski sums and network vulnerability analysis," *Discrete & Computational Geometry*, vol. 52, 2014.

[22] S. Har-Peled and B. Raichel, "On the complexity of randomly weighted multiplicative voronoi diagrams," *Discrete & Computational Geometry*, vol. 53, 2015.

[23] *PostgreSQL*, 2019, https://www.postgresql.org/.

[24] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu, "Prominent streak discovery in sequence data," in *SIGKDD*, 2011.

[25] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li, "Discovering general prominent streaks in sequence data," *TKDD*, vol. 8, 2014.

[26] B. Jiang and J. Pei, "Online interval skyline queries on time series," in *ICDE*, 2009.

[27] M. L. Lee, W. Hsu, L. Li, and W. H. Tok, "Consistent top-k queries over time," in *DASFAA*, 2009.

[28] H. Wang, Y. Cai, Y. Yang, S. Zhang, and N. Mamoulis, "Durable queries over historical time series," *TKDE*, vol. 26, 2014.

[29] J. Gao, P. K. Agarwal, and J. Yang, "Durable top-k queries on temporal data," *VLDB*, vol. 11, 2018.

[30] N. Mamoulis, K. Berberich, S. Bedathur *et al.*, "Durable top-k search in document archives," in *SIGMOD*, 2010.

[31] K. Semertzidis and E. Pitoura, "Durable graph pattern queries on historical graphs," in *ICDE*, 2016.

[32] K. Semertzidis *et al.*, "Top-k durable graph pattern queries on temporal graphs," *TKDE*, vol. 31, 2018.

[33] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *17th ICDE*. IEEE, 2001, pp. 421–430.

A. Implementation Details

For simplicity and usability, we adopt a more straightforward tree-based implementation that better serves our purpose for answering a preference top-k query in a time window.

Algorithm 4: Tree Index Construction

Input: Dataset P

Output: A Tree Index $\mathcal{T}$ for Preference Top-k Query

1 **Def** *BuildTree*(t_1, t_2)**:**

2   **if** $t_1 > t_2$ **then**

3   **return** null;

4   **else if** $t_1 == t_2$ **then**

5   create a leaf node n;

6    $n.\text{skyline} \leftarrow P[t_1]$;

7    $n.\text{interval} \leftarrow [t_1, t_2]$;

8   **return** n;

9   **else**

10   create a node n;

11    $t_m \leftarrow t_1 + (t_1 + t_2)/2$;

12    $n.\text{left\_child} \leftarrow \text{BuildTree}(t_1, t_m)$;

13    $n.\text{right\_child} \leftarrow \text{BuildTree}(t_m + 1, t_2)$;

14    $n.\text{interval} \leftarrow [t_1, t_2]$;

15    $n.\text{skyline} \leftarrow \mathbb{S}^1\big(\mathbb{S}^1\big(P([t_1, t_m])\big) \cup \mathbb{S}^1\big(P([t_m + 1, t_2])\big)\big)$;

16   **return** n;

Consider a query time W decomposed into n non-empty disjoint time intervals $W = \bigcup_{i=1}^{n} I_i$. Assume for each interval I_i we know the highest score (with respect to $\mathbf{u}$) among $P(I_i)$, referred to as *interval max score*. It is sufficient to use at most k out of n intervals [11] to answer a preference top-k query $Q(\mathbf{u}, k, W)$ if the chosen intervals have the k highest interval max scores. Based on this idea, our implementation takes advantages of two important properties of *skyline* [33] to improve the efficiency of index construction and query procedure.

As shown in Algorithm 4, the tree index is built upon the dimension of arriving time of all points in P in a bottom-up manner. Each leaf node corresponds to a single timestamp (Line 6) and each internal node represents a time window (Line 14). Each tree node also contains a skyline of points arriving during its window. Skylines in all internal nodes can be efficiently computed from bottom to up (Line 15).

Algorithm 5 specifies the query procedure using the tree index. Starting from the canonical intervals (nodes) of query window I (Line 4), we recursively split long intervals [12] into smaller ones (Line 10-13), and use a priority queue to

---

[11] Using all records in P that arrive during these k time intervals to compute the top-k results.

[12] The pre-determined value of LENGTH_THRESHOLD controls the granularity of the chosen k intervals for preference top-k computations. By default, we set LENGTH_THRESHOLD=128.

Algorithm 5: Preference Top-k Query $Q(\mathbf{u}, k, I)$

Input: P, $\mathcal{T}$, $\mathbf{u}$, k, and I
Output: $\pi_{\leq k}(\mathbf{u})I$

```
1  Def PreferenceTopK(I, u, k):
2      candidates ← ∅;
3      Q (priority queue in descending order of key)
          ← ∅;
4      N ← a set of canonical nodes from T that covers
          I;
5      for n_i ∈ N do
6          Compute interval max score v_i using
              n_i.skyline;
7          Q.push(v_i, n_i);
8      while |candidate| < k and !Q.empty() do
9          v, n ← Q.top(), Q.pop();
10         if |n.interval| > LENGTH_THRESHOLD then
11             n_l ← n.left_child, n_r ← n.right_child;
12             Compute v_l, v_r from n_l, n_r using skylines;
13             Q.push(v_l, n_l), Q.push(v_r, n_r);
14         else
15             candidate.push(n);
16     Compute π_≤k(u)I using candidates;
17     return π_≤k(u)I;
```

remember at most k intervals that have the highest interval max scores (Line 15). Finally, a preference top-k result is computed using at most k such intervals and all corresponding records in P (no more than $k*$LENGTH_THRESHOLD in total). We can efficiently compute the interval max score for any interval I (Line 6 and 12).

B. Missing Proofs of Section III

Proof of Lemma 1. Let $I = [a, b]$ and $\rho = [b - \tau, b]$. Let $S_\rho = S \cap \rho$, i.e., the set of durable records with timestamp in ρ. We show that the number of false checks in ρ is $O(|S_\rho| + k)$. Without loss of generality, assume that for any pair of records p_i, p_j with $i < j$, $p_i.t < p_j.t$.

We consider two types of false checks in ρ. If the algorithm finds a false check immediately after a durable record then this is a type-1 false check. Otherwise it is a type-2 false check. From the definition, the number of type-1 false checks is bounded by $O(|S_\rho|)$. Next we show that the number of type-2 false checks in ρ is bounded by $O(k)$. If the number of records in ρ is less than k then the result follows, so we assume that $|P[b - \tau, b]| > k$.

Recall that if the algorithm visits a record p it computes the top-k elements in $[p.t - \tau, p.t]$. Let U_p be the list of the top-k items in $[p.t - \tau, p.t]$. Let $Z_p = U_p \cap \rho$, be the list of these top-k elements that lie in ρ. Generally we refer to Z_p as a Z list. At the beginning of the algorithm assume that we find the top-k elements in a window of length τ from the rightmost item in ρ, so we have a list Z with $|Z| \leq k$. We show the following two observations. i) Each time that the algorithm

finds a type-2 false check the new Z list of top-k records in ρ has cardinality at least one less than the previous list. ii) The cardinalities of the Z lists as we run the algorithm in ρ are never increasing. If we show (i), (ii) we could argue that after the algorithm finds k type-2 false checks in ρ, the Z list will be empty and the algorithm will visits a record out of ρ.

Without loss of generality, assume that the rightmost record in ρ was a type-2 false check. Let Z_r be the current list as defined above. The algorithm visits the record with the largest timestamp in Z_r, say p, which is a type-2 false check. Let $Z_p = U_p \cap \rho$ be the new list. We compare the new list Z_p with the old list Z_r. Notice that every record $q \notin Z_r$ with time $q.t \in [b - \tau, p.t]$ has $f_\mathbf{u}(q) < f_\mathbf{u}(p)$ (1), otherwise Z_r would not be in the correct top-k list. Furthermore, p is a false check because there are at least k records in $[p.t - \tau, p.t)$ with score larger than the score of p, (2). From (1), (2) it follows that $Z_p \subset Z_r$. Hence, the cardinality of the new Z list is less than the cardinality of the previous Z list. In addition, notice that there are at least $k - |Z_p|$ records in $[p.t - \tau, b - \tau]$ with scores greater than the score of p, and generally greater than the score of any record in $P[b - \tau, p.t] \setminus Z_p$, (3).

In order to complete the proof we need to show what is the new Z list when the algorithm visits a series of durable records. Assume that Z_p is the current list (or the initial one) and the algorithm visits Z_p's record with the larger timestamp. Assume that the algorithm finds a series of durable records, where j of them belong in Z_p. Notice that $j \geq 1$. Let q be the type-1 false check that the algorithm visits (after the series of durable records) and let Z_q be the new list. We need to show that $|Z_q| \leq |Z_p|$. We assume that $q \notin Z_p$ (if $q \in Z_p$ then notice that $Z_q \subset Z_p$ so the result follows). Recall from (3) that there are at least $k - |Z_p|$ records with timestamp $[p.t - \tau > q.t - \tau, b - \tau]$ and with score greater than the score of q. We call these records A. Moreover, there are $|Z_p| - j$ records in Z_p with timestamp in $[b - \tau, q.t)$ and with score greater than the score of q. We call these records B. We have $|Z_q| \leq |B| + (k - |A| - |B|) = k - |A| \leq |Z_p|$. Hence, we conclude that there are $O(k)$ type-2 false checks and the total number of false checks in ρ is $O(|S_\rho| + k)$.

There are $\left\lceil \frac{|I|}{\tau} \right\rceil$ intervals of length τ in I so the total number of false checks is $O(|S| + k \left\lceil \frac{|I|}{\tau} \right\rceil)$. □

C. Missing Proofs of Section IV

We first introduce some useful notation. Let $dens(t)$ be the density of a timestamp t, i.e., the number of blocking intervals that contain t. Notice that $dens(t)$ is changing as we execute the algorithm. If a record p_i is blocked by at least k records, i.e., $dens(p_i.t) \geq k$, at line 7 of Algorithm 3 then we call it an *auxiliary record*. Overall, we have that a record can be a durable record, a false check (we run a top-k query but the record does not belong in the solution), or an auxiliary record.

We first start with a lemma that will be useful later.

Lemma 6. *Let M_i be a set that is empty after the algorithm considering a (auxiliary) record from M_i with density at least k, and let $[l_i, r_i]$ be its corresponding sub-interval. Then one of*

the two cases hold: The density of each timestamp in $[l_i, r_i]$ is at least k or the algorithm has visited all records in $P([l_i, r_i])$.

Proof. If $|P([l_i, r_i])| \leq k$ then the algorithm visits all records in $P([l_i, r_i])$, since we always consider the top-k records in $[l_i, r_i]$. If $|P([l_i, r_i])| > k$ then we show that when M_i is empty every timestamp in $[l_i, r_i]$ has density at least k.

We prove the following argument by induction: When the algorithm visits a new auxiliary record p_j in a set M_j then any timestamp in $[l_j, t_j]$ has density at least k. Let p_1 be the first auxiliary record that the algorithm finds and let M_{i_1} be the set that it belongs to. Since p_1 is an auxiliary record we have that $dens(p_1.t) \geq k$ at the moment we visit p_1. Furthermore, notice that the algorithm did not consider any other record in $[l_{i_1}, p_1.t]$ in a previous iteration so we can argue that the density of every record in $[l_{i_1}, p_1.t]$ is at least k. In addition, notice that it is not possible to find any durable record or any false check in $[l_{i_1}, p_1.t]$ in the future. As a result, if we visit p_1 again in the future it will be an auxiliary record in a set with left endpoint the same l_{i_1} timestamp. Let p_{h-1} be an auxiliary record that the algorithm visits in set $M_{i_{h-1}}$ and let assume that any record in $[l_{i_{h-1}}, p_{h-1}.t]$ has density at least k. Let p_h be the next auxiliary record that the algorithm visits and let assume that it belongs in a set M_{i_h}. First assume that the algorithm has visited p_h in a previous iteration. Let M_f be the set that contained p_h when the algorithm first visited p_h. At the moment when the algorithm first visited p_h, we had that $dens(p_h.t) \geq k$ and from the induction hypothesis we have that every timestamp in $[l_f, p_h.t]$ had density at least k. Hence, there was no other durable record or false check in $[l_f, p_h.t]$ in the future. That means that $l_f = l_{i_h}$ and so it holds that every record in $[l_{i_h}, p_h.t]$ has density at least k. Next, assume that this is the first time that we visit the auxiliary record p_h. If this is the first auxiliary record in M_{i_h} we have that the density of every record in $[l_{i_h}, p_h.t]$ has density at least k because $dens(p_h.t) \geq k$ and there is no subinterval that starts in $[l_{i_h}, p_h.t]$. Then, we study the case where p_h is not the first auxiliary record that the algorithm finds in set M_{i_h}. Let p_u be the auxiliary record in M_{i_h} with the largest timestamp just before the algorithm found p_h. From induction hypothesis we know that the density of every record in $[l_{i_h}, p_u.t]$ is at least k. If $p_h.t \leq p_u.t$ then $[l_{i_h}, p_h.t] \subseteq [l_{i_h}, p_u.t]$ so any record in $[l_{i_h}, p_h.t]$ has density at least k. The last case to consider is when $p_h.t > p_u.t$. Since $dens(p_h.t) \geq k$, and since there is no sub-inerval that starts in $(l_u, p_h.t)$ we have that every record in $[l_u, p_h.t]$ has density at least k. We conclude that the density of every timestamp in $[l_{i_h}, p_h.t]$ is at least k.

Now we are ready to prove our lemma. If $|P([l_i, r_i])| > k$ and M_i is empty it means that the algorithm has already considered k auxiliary records in $[l_i, r_i]$. Let p_u be the auxiliary record in M_i with the largest timestamp. From the induction we have that the density of every record in $[l_i, p_u.t]$ is at least k. Furthermore, the algorithm has visited k auxiliary records and hence it has added at least k blocking intervals with left endpoint in $[l_i, p_u.t]$. All the intervals we add have length τ and $r_i - l_i \leq \tau$ so all timestamps in the interval $[p_u.t, r_i]$ have

density at least k. We conclude that the density of each record in $[l_i, r_i]$ is at least k. □

Proof of Lemma 2. Let S^* be the durable records in I. We show that $S \subseteq S^*$ and $S^* \subseteq S$ showing that $S = S^*$. The algorithm always checks by running a top-k query if a record should be in the solution (line 8 of Algorithm 3) so $S \subseteq S^*$.

Next we show the other direction. The algorithm visits the records in descending (on score) order so it is not possible that a record $p \in S^*$ is blocked by at least k records before the algorithm visits p. Before we argue that $S^* \subseteq S$ we also need to make sure that the algorithm does not miss any durable record in a sub-interval $[l_j, r_j]$ that corresponds to an empty set M_j. In Lemma 6 we showed that all timestamps in $[l_j, r_j]$ have density at least k so there is no additional durable record in this sub-interval. Hence $S^* \subseteq S$, and overall we conclude that $S = S^*$. □

Let p_i be a false check that the algorithm just found, and let P_i' be the top-k records in $[p_i.t - \tau, p_i.t)$, as we had in the algorithm. Let p_i' be the record in P_i' with the largest timestamp. We say that p_i is *assigned* to p_i'. If $p_i.t' < a$, where a is the timestamp such that $I = [a, b]$, then p_i is assigned to a. The next lemma follows from the definition.

Lemma 7. *Assume that the algorithm just found the false check p_i. After adding all the blocking intervals from P_i' we have that the density of every timestamp in $[p_i'.t, p_i.t]$ is at least k.*

We show the next lemma which is useful to bound the number of false checks.

Lemma 8. *Let p_i be a false check and p_i' be the record that it is assigned to. Before adding the k blocking intervals from all records in P_i' (as defined above) we have that either $dens(p_i'.t) \geq k$, or $p_i' \in S$ and $dens(p_i'.t) < k$, or $p_i' = a$.*

Proof. If $p_i'.t < a$ then from the definition p_i' is a. (Notice that if we find more than one false checks that are assigned to a then $dens(a) > k$, so this case can be considered the same as $dens(p_i'.t) > k$.)

Next, we assume that $p_i'.t \geq a$. We prove the lemma by contradiction. Let p_i' be a record that does not belong in S and $dens(p_i'.t) < k$. Notice that $f_\mathbf{u}(p_i') > f_\mathbf{u}(p_i)$. Since p_i' is not in S it can be either: a false check, an auxiliary record, or a record that the algorithm has not visited before. If p_i' is a false check then from Lemma 7 we have that $dens(p_i'.t) \geq k$ at the moment that we found p_i' for first time, which is a contradiction. If p_i' is an auxiliary record then from Lemma 6 we have that $dens(p_i'.t) \geq k$, which is a contradiction. If p_i' is a record that the algorithm has not considered before then there are two cases: a) p_i' belongs in an interval $[l_j, r_j]$ of a set M_j that we have removed from M because we have already visited its top-k records. From Lemma 6 we know that $dens(p_i'.t) \geq k$, which is a contradiction. b) p_i' belongs in an interval $[l_j, r_j]$ of a set M_j that there still exists in H. Since $f_\mathbf{u}(p_i') > f_\mathbf{u}(p_i)$ it means that p_i is not the record with

the highest score among the sub-intervals that are not removed from M, which is a contradiction.

In any case we proved that either p_i' has density at least k, or p_i' has density less than k and $p_i' \in S$, or $p_i' = a$. $\quad\square$

Proof of Lemma 3. If a false check p_i is assigned to a durable record with density less than k then we call it type-1 false check. Otherwise, it is a type-2 false check.

Let p_i be a type-1 false check so we have that $p_i' \in S$ and $dens(p_i'.t) < k$. After adding all the k segments from P_i' we have that $dens(p_i'.t) \geq k$. The next time that p_i' will be assigned by another false check the density of p_i' will be at least k so it will be a type-2 false check. Hence, it is straightforward to bound the number of type-1 false checks, which is at most $O(|S|)$.

Next we focus on type-2 false checks. Let $[l, r]$ be one of the initial disjoint τ-length windows from line 2 of Algorithm 3. We show that after finding k type-2 false checks in $[l, r]$ the density of all timestamps in $[l, r]$ is at least k. If that is the case then the algorithm will not find any other false check in $[l, r]$.

Let t be any timestamp in $[l, r]$. We show that $dens(t) \geq k$ after finding k type-2 false checks in $[l, r]$. If one of the false checks in $[l, r]$ lies on t then we already have that $dens(t) \geq k$. Let assume that the algorithm finds k_1 type-2 false checks in $[l, t)$ and k_2 type-2 false checks in $(t, r]$, where $k_1 + k_2 = k$. If $k_1 \geq k$ then $dens(t) \geq k$, so the interesting case is when $k_1 < k$ and $k_2 \geq 1$. Let χ be the total number of blocking intervals that the algorithm has added having their right-endpoint in $[t, r]$ after finding all the k type-2 false checks in $[l, r]$, and let X be the set of those intervals. We have that $dens(t) \geq k_1 + \chi$. We show that $k_1 + \chi \geq k$ or equivalently $k_2 \leq \chi$.

Let p_i be a type-2 false check that the algorithm just found in $(t, r]$. Let p_i' be the record that p_i is assigned to, as we defined above. If $p_i'.t \leq t$ then we immediately have that $dens(t) \geq k$ after adding the at most k new segments from the set P_i' (Lemma 7), so this case is not interesting. (Notice that if $p_i'.t < a$, before p_i' is set to be a, then this is always the case since $t \geq a$).

Now, we assume that for each p_i which is a type-2 false check in $(t, r]$, it holds that $p_i'.t \in (t, p_i.t)$. The main idea to prove that $k_2 \leq \chi$ is the following: Each time that the algorithm finds a type-2 false check in $(t, r]$ we find an unmarked interval in X and we mark it. In particular, we show that there always be such an unmarked segment in X with its right endpoint in $[p_i'.t, p_i.t)$. Since p_i is a type-2 false check we have that $dens(p_i'.t) \geq k$ and $dens(p_i.t) < k$, at the moment that the algorithm visits p_i (before adding the at most k segments from P_i'). Let Z_1 be the current blocking intervals with right endpoint in $[p_i'.t, p_i.t)$ and $z_1 = |Z_1|$. Let Z_2 be the current blocking intervals with left endpoint in $(p_i'.t, p_i.t]$, and $z_2 = |Z_2|$. Let B be the current blocking intervals with left endpoint in $[p_i.t - \tau, p_i.t]$. We have that $dens(p_i.t) < k \Leftrightarrow |B| < k$, (1). We also have $dens(p_i'.t) \geq k \Leftrightarrow |B| - z_2 + z_1 \geq k$, (2). From (1), (2), we have that $z_1 > z_2 \Leftrightarrow z_1 \geq z_2 + 1$. By definition, notice that the false

checks with time instance $\leq p_i'.t$ cannot mark a segment in Z_1. Furthermore, a previous false check with timestamp at the right of $p_i.t$ cannot mark a segment in Z_1: Let p_j be a false check that the algorithm found in a previous iteration in $(p_i.t, r]$ and let p_j' be the record that it is assigned to. If $p_j'.t > p_i.t$ then the marking process does not mark any segment in Z_1. Otherwise, if $p_j'.t \leq p_i.t$ then the density of all records in $[p_j'.t, p_i.t] \cup [p_i.t, p_j'.t]$ would be at least k after the algorithm adds the segments from P_j', which is a contradiction because $dens(p_i.t) < k$ when we visit p_i. Hence only false checks in $(p_i'.t, p_i.t]$ can mark segments in Z_1. Recall that Z_2 are the current segments with left endpoints in $(p_i'.t, p_i.t]$. Even if all segments in Z_2 were created by type-2 false checks and even if all of them mark segments from Z_1, we showed that $z_1 \geq z_2 + 1$, so we can always find a new unmarked segment in Z_1. Notice that any segment in Z_1 has its right endpoint in $[p_i'.t, p_i.t)$ and since all the segments have length τ, they contain t and hence they belong in X. Each time that we find a type-2 false check in $(t, r]$ we mark a new segment in X, so $k_2 \leq \chi$ and we conclude that $dens(t) \geq k$.

Recall that t can be any record in $[l, r]$, so we showed that after finding k type-2 false checks in $[l, r]$ the density of every timestamp in $[l, r]$ is at least k. As a result, the algorithm will not find any other false check in $[l, r]$. There are at most $\lceil \frac{|I|}{\tau} \rceil$ disjoint τ-length windows in I so the number of type-2 false checks is bounded by $O(k \lceil \frac{|I|}{\tau} \rceil)$ The overall number of false checks along with the durable records is $O(|S| + k \lceil \frac{|I|}{\tau} \rceil)$. $\quad\square$

D. Missing Proofs of Section V

Proof of Lemma 5. We show the result extending the main ideas from [15]. Let $P(I) = \{p_{j+1}, \ldots, p_{j+L}\}$. For $p_i \in P(I)$, let X_i be a random variable which is 1 if $p_i \in \mathcal{C}$, and 0 otherwise. From linearity of expectation we have that $\mathbf{E}[|C|] = \mathbf{E}\left[\sum_{i=j+1}^{j+|I|} X_i\right] = \sum_{i=j+1}^{j+|I|} \mathbf{E}[X_i] = \sum_{i=j+1}^{j+|I|} \mathbf{Pr}[X_i = 1]$. We focus on computing $\mathbf{Pr}[X_i = 1]$. Let $P_i = P([p_{i-\tau}.t, p_i.t]) = \{p_{i-\tau}, \ldots, p_{i-1}, p_i\}$. By independence we have that the probability of each point in P_i to be in the k-skyband of P_i is the same, so we can compute $\mathbf{Pr}[X_i = 1]$ by first finding the expected size of the k-skyband in P_i and then divide it by the number of points, $\tau + 1$.

Let B_i be the k-skyband of the $\tau + 1$ points P_i. Let $V_j \subset N$ for $1 \leq j \leq d$, with $|V_j| = \tau + 1$ such that V_j contains the values that are assigned to the j-th coordinate of the points in P_i. We compute $\mathbf{E}[|B_i| \mid V_1, \ldots, V_d]$. Let $A(\tau + 1, d)$ be the expected size of the k-skyband of a set $\bar{P}$ with $\tau + 1$ points in $\mathbb{R}^d$ in the d-dimensional random permutation model. Notice that $A(\tau + 1, d) = \mathbf{E}[|B_i| \mid V_1, \ldots, V_d]$. We compute $A(\tau + 1, d)$ as follows. From linearity of expectation we can compute the probability that a point in $\bar{P}$ belongs in the k-skyband and take the sum of them, $A(\tau + 1, d) = \sum_{\bar{p} \in \bar{P}} \mathbf{Pr}[\bar{p} \in k\text{-skyband of } \bar{P}]$. Assume that a point $\bar{p} \in \bar{P}$ has the g-th largest first coordinate among the points in $\bar{P}$. Notice that this can happen with probability $\frac{1}{\tau+1}$. Since the first coordinate of the g-th point ($\bar{p}$) is greater than the first

coordinates of $g-1$ points it cannot be dominated by any of those. Therefore, the g-th point belongs in the k-skyband if and only if its remaining $d-1$ coordinates belong in the k-skyband among the points in $\bar{P}$ with the g-th through the $(\tau+1)$-th largest first coordinate. The probability that the g-th point is in the k-skyband is, by independence, the expected number of the k-skyband in the remaining points and coordinates, which is $A(\tau+1-g+1, d-1)$, divided by the total number of the remaining points in the set which are $\tau+1-g+1$. Notice that $A(k', y) = k'$ for $k' \leq k$ and any y. Hence, we have $A(\tau+1, d) = \sum_{j=1}^{\tau+1} \sum_{g=1}^{\tau+1} \frac{1}{\tau+1} \frac{A(\tau+1-g+1, d-1)}{\tau+1-g+1} = \frac{1}{\tau+1} \sum_{j=1}^{\tau+1} \sum_{J=1}^{\tau+1} \frac{A(J, d-1)}{J} = \sum_{J=1}^{\tau+1} \frac{A(J, d-1)}{J}$. Notice that $A(x, y)$ is monotonically increasing in x, so if $x_1 \leq x_2$, then $A(x_1, y) \leq A(x_2, y)$ for any y. Furthermore, we note that $A(\tau+1, 1) = k$ since in one dimension the top-k points belong in the k-skyband. We have, $A(\tau+1, d) = \sum_{J=1}^{\tau+1} \frac{A(J, d-1)}{J} \leq A(\tau+1, d-1) \sum_{J=1}^{\tau+1} \frac{1}{J} \leq A(\tau+1, d-1) O(\log \tau)$. Iterating this recurrence on d until $A(\tau+1, 1) = k$ gives the upper bound $A(\tau+1, d) = O(k \log^{d-1} \tau)$.

We conclude that $\mathbf{E}\left[|B_i| \mid V_1, \ldots, V_d\right] = O(k \log^{d-1} \tau)$. Notice that $\mathbf{Pr}\left[V_1, \ldots, V_d\right] = \frac{1}{\binom{n}{\tau+1}^d}$ and all possible sets of $V_1, \ldots, V_d$ are $\binom{n}{\tau+1}^d$ so we have that $\mathbf{E}\left[|B_i|\right] = O(k \log^{d-1} \tau)$, and $\mathbf{Pr}\left[X_i = 1\right] \sim \frac{O(k \log^{d-1} \tau)}{\tau+1}$. Overall we conclude that $\mathbf{E}\left[|\mathcal{C}|\right] = \sum_{i=j+1}^{j+|I|} \mathbf{Pr}\left[X_i = 1\right] = O(\frac{k|I|}{\tau} \log^{d-1} \tau)$. $\qquad\square$