

# Durable Top-K Instant-Stamped Temporal Records with User-Specified Scoring Functions

Junyang Gao<sup>§</sup>  
Google Inc.

Stavros Sintos<sup>§</sup>  
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

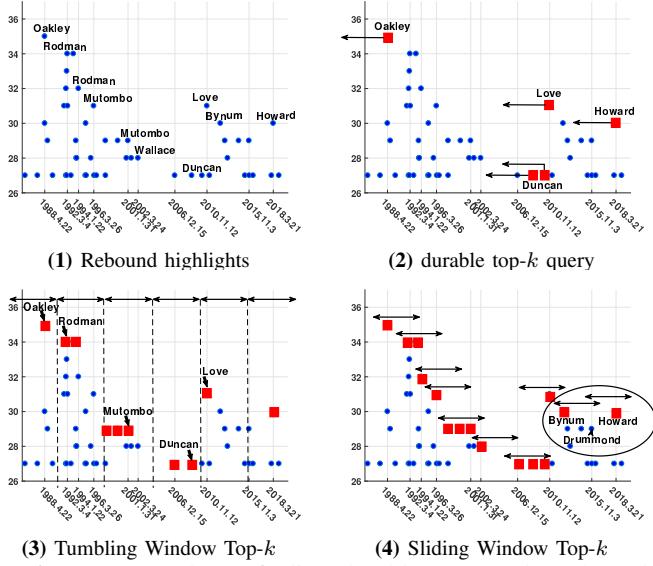
and marketing, because people are naturally attracted by those events that “stood the test of time.” Such analysis of durability is a useful part of the toolbox for anybody who works with historical data, and can be particularly helpful to journalists and marketers in identifying newsworthy facts and 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  $\tau$ , records that are within top  $k$  during a  $\tau$ -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 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

<sup>§</sup>Most of the work was conducted when authors were at Duke University.



**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.

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, 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 Mid-western 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 sensitive 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<sup>1</sup>.
- 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  $k$ ,  $\tau$  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.

<sup>1</sup>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
$\tau$	Durability duration
$u$	Query vector
$s(n), q(n)$	Space, query time of top- $k$ index

- 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.

## 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  $(p.x_1, p.x_2, \dots, p.x_d) \in \mathbb{R}^d$ . For simplicity, we consider a discrete time domain of interest  $\mathbb{T} = \{1, 2, \dots, n\}$ , and let  $p.t \in \mathbb{T}$  denote the *arrival time* of  $p$ . All records in  $P$  are organized by increasing order of their 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,  $f : \mathbb{R}^d \rightarrow \mathbb{R}$ . 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  $\tau$ -durable<sup>2</sup> if  $p \in \pi_{\leq k}([p.t - \tau, p.t])$ . That is,  $p$  remains in the top- $k$  for  $\tau$  time during  $[p.t - \tau, p.t]$ . Note that if a record  $p$  is  $\tau$ -durable, then it is also  $\tau'$ -durable for  $\tau' \leq \tau$ . We are interested in finding records with “long enough” durability, i.e., durability at least  $\tau$ . Given a query interval  $I$  and a durability threshold  $\tau \in [1, |\mathbb{T}|]$ , a *durable top- $k$  query*, denoted  $\text{DurTop}(k, I, \tau)$ , returns the set of  $\tau$ -durable records that arrive during  $I$ ; i.e.,  $\text{DurTop}(k, I, \tau) = \{p \in P(I) \mid p \in \pi_{\leq k}([p.t - \tau, p.t])\}$ . 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

<sup>2</sup>If  $\tau$  is obvious from the context, we drop  $\tau$  from the definition, i.e., we say that a record is durable.

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.

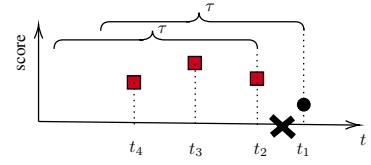
Here, we give some concrete examples of  $f$  that are widely used in real-life applications, for which efficient top- $k$  query building blocks exist. Consider the following class of scoring functions parameterized by  $\mathbf{u}$ , which captures user preference:

- *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].

The above three forms of the preference functions are particularly desirable 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. As mentioned earlier, users can replace the scoring function 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, through the per-invocation complexity of the top- $k$  query building block.

**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 incre-



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

mental top- $k$  maintenance over sliding windows<sup>3</sup>. 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.

### III. TIME-PRIORITIZED APPROACH

The time-prioritized approach is straightforward: we visit records in time order and check their durability sequentially. The standard sliding-window technique [11], which we refer to as *Time-Base* or *T-Base*, is in this category. To beat the linear complexity of T-Base, we propose a new algorithm (Section III-A) using the observation that we can skip unpromising records as we discover durable ones. What is surprising is how this simple observation leads to provably substantial reduction in complexity (Sections III-B and V).

#### A. 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, T-Base (and its many related variants) does not best serve our needs. Since the entire data is available in advance, continuously sliding the window wastes too much time on those non-durable records. After all, meaningful durable top- $k$  queries tend to 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  $\tau$ -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  $\tau$ -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); see Algorithm 1 for pseudocode. 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.

<sup>3</sup>In particular, the idea of Skyband Maintenance Algorithm (SMA) to reduce the number of top- $k$  re-computations from scratch.

---

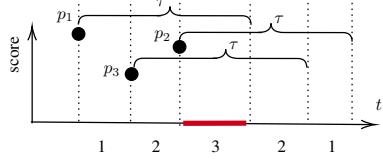
**Algorithm 1:** T-Hop ( $k, I, \tau$ )

---

**Input:**  $P, k, \tau$ , and  $I : [t_1, t_2]$ .  
**Output:**  $\text{DurTop}(k, I, \tau)$

- 1 Initialize answer set:  $S \leftarrow \emptyset$ , top- $k$  set:  $\pi_{\leq k} \leftarrow \emptyset$ ;
- 2  $t_{curr} \leftarrow t_2$ ;
- 3 **while**  $t_{curr} \geq t_1$  **do**
- 4      $\pi_{\leq k} \leftarrow Q(k, [t_{curr} - \tau, t_{curr}])$ ;
- 5     **if**  $P[t_{curr}] \in \pi_{\leq k}$  **then**
- 6          $S \leftarrow S \cup P[t_{curr}]$ ;  $t_{curr} \leftarrow t_{curr} - 1$ ;
- 7     **else**  $t_{curr} \leftarrow$  most recent arrival time of records in  $\pi_{\leq k}$ ;
- 8 **return**  $S$ ;

---



**Fig. 3:** Blocking mechanism in score-prioritized approach

#### B. Time Complexity Analysis of T-Hop

The time complexity of T-Hop 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. Due to the space limits we provide all proofs in the extended version of the paper [14].

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

Overall, with an efficient top- $k$  module, T-Hop answers a durable top- $k$  query  $\text{DurTop}(k, I, \tau)$  in  $O((|S| + k \lceil \frac{|I|}{\tau} \rceil)(q(n) + k) \log n)$  time. Compared to T-Base, T-Hop runs in sublinear query time (assuming that the ratio  $\lceil \frac{|I|}{\tau} \rceil$  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. Using the results in [1]–[6], our index can be implemented with 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  $k \lceil \frac{|I|}{\tau} \rceil$ . In the worst case, the term  $k \lceil \frac{|I|}{\tau} \rceil$  can be arbitrarily large compared 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  $O(k \lceil \frac{|I|}{\tau} \rceil)$ , meaning that the expected number of top- $k$  queries executed by the algorithm is  $O(|S|)$ .

#### IV. SCORE-PRIORITIZED APPROACH

One potential 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  $O(|S| + k \lceil \frac{|I|}{\tau} \rceil)$  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

---

**Algorithm 2:** S-Band ( $k, I, \tau$ )

---

**Input:**  $P, k, \tau$ , and  $I$ .  
**Output:**  $\text{DurTop}(k, I, \tau)$

- 1  $S \leftarrow \emptyset, \Gamma \leftarrow \emptyset$ ;
- 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  $\Gamma$  **then**
- 6          $\pi_{\leq k} \leftarrow Q(k, [p.t - \tau, p.t])$ ;
- 7         **if**  $p \in \pi_{\leq k}$  **then**  $S \leftarrow S \cup \{p\}$ ;
- 8         **else**
- 9             **for**  $q \in \pi_{\leq k} \wedge q$  not visited before **do**
- 10                  $\Gamma \leftarrow \Gamma \cup \{[q.t, q.t + \tau]\}$ ;
- 11      $\Gamma \leftarrow \Gamma \cup \{[p.t, p.t + \tau]\}$ ;
- 12 **return**  $S$ ;

---

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  $\tau$  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  $\tau$ -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  $\tau$ -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  $\tau$ -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.

A straightforward score-prioritized approach, which we refer to as *Score-Base* (or, S-Base), is to first sort all records in the query interval according to the scoring function  $f$  and then traverse records in descending order of scores together with blocking mechanism. But the sorting overhead would become the bottleneck of query efficiency, incurring cost super-linear (i.e.,  $n \log n$ ) in the data size. We describe two better algorithms to alleviate the expensive sorting component in the following sections. They differ on how high-score records are found and how blocking intervals are maintained.

### A. Score-Band Algorithm (Monotone $f$ 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.,  $S \subseteq \mathcal{C}$ , then we could get a faster algorithm by only sorting  $\mathcal{C}$ , instead of all records in the query interval. It is well-known that the  $k$  records with the highest score, with respect to *any monotone* scoring functions, belong to the  $k$ -skyband.<sup>4</sup> Hence, if a record  $p$  is  $\tau$ -durable for a top- $k$  query (with respect to a monotone  $f$ ), then  $p$  must also be  $\tau$ -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  $\tau_p$  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 [15] (or kd-tree, R-tree in practice). To answer  $\text{DurTop}(k, I, \tau)$ , 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  $O(n)$  space and the query time is  $O(|\mathcal{C}| + \log n)$  in order to get the set  $\mathcal{C}$ .

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  $\kappa_0$  then we can construct  $\kappa_0$  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  $k = 2^0, 2^1, \dots, 2^{\log n}$ , 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  $\tau$ -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  $k, I, \tau$ , 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  $\tau$ -durable, we add  $p$  to answer set. Otherwise,

<sup>4</sup>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  $k = 1$ .

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  $\tau$ -durable (Line 6). This 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.

**Query Time 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  $O(|\mathcal{C}|(q(n) + k) \log n)$ , 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 [16] (where the coordinates of records 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 need to sort all records in  $\mathcal{C}$ ; this might make S-Band much slower due to the size of  $\mathcal{C}$  that increases (in expectation) exponentially on the dimension  $d$ .

### B. Score-Hop Algorithm

The data reduction strategy of S-Band offers adequate benefits for improving the overall running time on datasets in low dimensions ( $\leq 5$ ). 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

---

**Algorithm 3: S-Hop ( $k, I, \tau$ )**


---

```

Input:  $P, k, \tau$ , and  $I : [a, b]$ .
Output:  $\text{DurTop}(k, I, \tau)$ 
1  $H \leftarrow \emptyset, S \leftarrow \emptyset, \Gamma \leftarrow \emptyset;$ 
2 for  $[l_i, r_i] : \text{disjoint } \tau\text{-length intervals in } I$  do
3    $M_i \leftarrow Q(\mathbf{u}, k, [l_r, r_i]);$ 
4    $H.\text{push}(M_i.\text{pop}());$ 
5 while  $H \neq \emptyset$  do
6    $p \leftarrow H.\text{pop}()$ , and let  $p \in M_j$ ;
7   if  $p$  lies in  $< k$  blocking intervals in  $\Gamma$  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  $q \in \pi_{\leq k} \wedge q \text{ not visited before}$  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        $H.\text{push}(M_j^-.\text{top}()), H.\text{push}(M_j^+.\text{top}());$ 
17     else if  $M_j \neq \emptyset$  then
18        $H.\text{push}(M_j.\text{pop}());$ 
19     if  $p$  not visited before then
20        $\Gamma \leftarrow \Gamma \cup \{[p.t, p.t + \tau]\};$ 
21 return  $S$ ;

```

---

$k$  blocking intervals. Let  $p$  be such a record. We run a durable top- $k$  query so if  $p$  is a  $\tau$ -durable record we add it in  $S$ . If  $p$  is not a  $\tau$ -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 interval  $I = [a, b]$ , we partition it into a set of  $\tau$ -length sub-intervals:  $[a, a + \tau], [a + \tau, a + 2\tau], \dots, [a + \lfloor \frac{|I|}{\tau} \rfloor \tau, b]$ . Let  $[l_i, r_i]$  be the  $i$ -th sub-interval, and in each interval we find the  $k$  records 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  $p$ ). 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.

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 ( $\geq 10$ ) datasets.

We prove the correctness of Score-Hop algorithm in the extended version of the paper [14].

### C. Time Complexity Analysis of S-Hop

The time complexity analysis of the S-Hop query procedure 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.

**Lemma 2.** *The total number of top- $k$  queries performed by the Score-Hop algorithm is  $O(|S| + k \lceil \frac{|I|}{\tau} \rceil)$ .*

Using Lemma 2, it can be shown that S-Hop answers a durable top- $k$  query in  $O((|S| + k \lceil \frac{|I|}{\tau} \rceil)(q(n) + k) \log n)$  time (with an efficient top- $k$  query procedure in  $O(q(n) + k)$ ). Similarly to T-Hop our index for S-Hop has near linear space and supports updates in polylogarithmic time [1]–[6].

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 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, and our best solutions, T-Hop and S-Hop, share 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 chosen *arbitrarily* (possibly by a powerful adversary with the advance knowledge of the query parameters) but their arrival order is random. This result essentially establishes that, under this model, the expected running time of T-Hop and S-Hop is 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  $\tau$ -durable  $k$ -skyband candidate set  $\mathcal{C}$  using the same probabilistic model used in [16].

### 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  $\mathbf{X} = x_1 < x_2 < \dots < x_n$  be a sequence of  $n$  arbitrary non-negative numbers chosen by an adversary, and let  $\sigma$  be a permutation of  $\{1, \dots, n\}$ . 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  $\sigma$ . As argued in [17], 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 [18]–[20], discrete geometry [21]–[23], and query processing [17]. Our main result is the following.

**Lemma 3.** *In the random permutation model, given  $k, \tau$  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  $\tau$ -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  $P_i^\tau = \{p_{i-\tau}, \dots, p_{i-1}\}$ . 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  $\bar{Q} = P_i^\tau \setminus Q$  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  $\mathbf{Pr}[A_Q | V]$  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  $\bar{Q}$  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  $\bar{Q}$ . Under such assignment, any permutations of values in  $Q$  and  $\bar{Q}$  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

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

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

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

**TABLE II:** Dataset summary

Dataset	Dimensionality	Size (# records)
<b>NBA-X</b>	1,2,3,5	1M
<b>Network-X</b>	2,3,5,10,20,30,37	5M
<b>Syn-X</b>	2	1M,2M,5M,10M,20M,50M,500M

$$\text{and finally, } \mathbf{E}[|S|] = \sum_i \mathbf{Pr}[X_i = 1] = k \frac{|I|}{\tau+1}. \quad \square$$

Combining Lemma 3 with the analysis of Sections III-B and IV-C, 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  $O(k \lceil \frac{|I|}{\tau} \rceil (q(n) + k) \log n)$ , where  $O(q(n) + k)$  reflects the time complexity of answering a top- $k$  query.

### B. Expected size of durable $k$ -skyband

In this subsection we bound the expected size of  $\tau$ -durable  $k$ -skyband records, denoted by  $\mathcal{C}$ , from Section IV-A 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  $P = \{p_1, \dots, p_n\}$  with  $p_i.t = i$ . We use the same random model as in [16] where (the attributes of) records are randomly generated.

**Lemma 4.** *If the attributes of records are generated randomly, then given  $k, \tau$  and  $I$ , we have  $\mathbf{E}[|\mathcal{C}|] = O(k \frac{|I|}{\tau} \log^{d-1} \tau)$ .*

Combining Lemma 4 with the analysis of Section IV-A, we get that the *expected* query time complexity of the Score-Band algorithm is  $O(k \lceil \frac{|I|}{\tau} \rceil (q(n) + k) \log n \log^{d-1} \tau)$ . 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 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

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

**NBA**<sup>5</sup> contains the performance of *each* NBA player in *each* game from 1983 to 2019, totaling  $\sim 1$  million 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**.

**Network**<sup>6</sup> is the dataset from KDD Cup 1999, containing  $\sim 5$  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

<sup>5</sup>NBA datasets were collected from <https://www.basketball-reference.com/>

<sup>6</sup><https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>

**TABLE III:** Query Parameters (default value in bold)

Parameter	Range
$k$	5, <b>10</b> , 15, 20, 25, 30, 35, 40, 45, 50
$\tau$	1%, 5%, 10%, 15%, <b>20%</b> , 25%, 30%, 40%, 50%
$ I $	10%, 20%, 30%, 40%, <b>50%</b> , 60%, 70%, 80%
$d$	1, 2, 3, 5, 10, 20, 30, 37

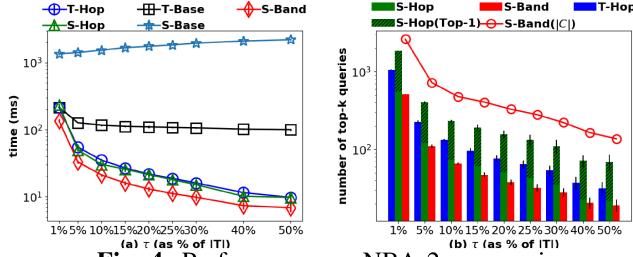


Fig. 4: Performance on NBA-2 as  $\tau$  varies.

to rank connections in order to identify unusual and potentially malicious ones. Records have unique timestamps and are ordered by these timestamps. Since the 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 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. The full size of Syn is 500 million (take nearly 30 Gigabytes disk space) and each data point has an unique arriving time. We further choose several subsets of Syn with 1, 2, 5, 10, 20 and 50 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  $\tau$  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- $k$

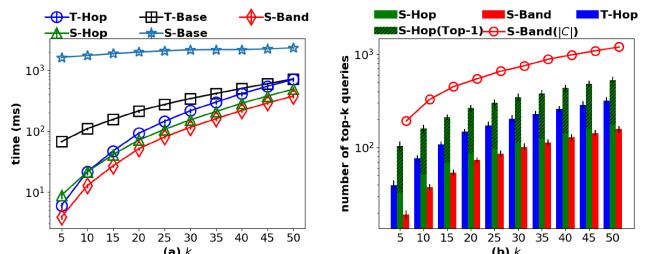


Fig. 5: Performance on NBA-2 as  $k$  varies.

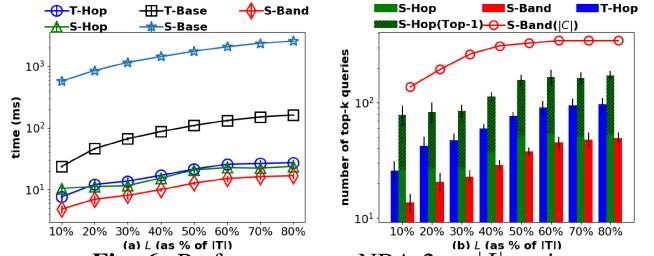


Fig. 6: Performance on NBA-2 as  $|I|$  varies.

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. Full description of the index construction and top- $k$  query procedure can be found in the extended version of this paper [14]. Our code is also available at [24]. 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), **T-Hop** (Section III-A), **S-Base** (Section IV), **S-Band** (Section IV-A) and **S-Hop** (Section IV-B). 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.

#### A. Algorithm Evaluations

According to the theoretical analysis of our algorithms in previous sections, the query efficiency depends on the length of durability window  $\tau$ , 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  $\tau$ .** In Figure 4, we investigate the performance of all durable top- $k$  solutions, as we vary durability  $\tau$ . Figure 4a 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  $\tau$ . All the rest solutions, T-Hop, S-Hop and S-Band, become more efficient as we increase  $\tau$ , 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 ( $\tau$  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 results can be found on Network-2 dataset. Due to space limit, please refer to the extended version of this paper [14] for more experiments.

Next, we take a closer look at T-Hop, S-Hop and S-Band in Figure 4b, 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  $\tau$  is large is that fewer top- $k$  queries are needed. A more selective query with larger  $\tau$  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. Again, similar trends can be found in the Network-2 dataset.

**Comparison of Algorithms when Varying  $k$ .** Next, we study the effect of  $k$  on efficiency. Results are shown in Figure 5. When we increase  $k$ , not only need we ask more top- $k$  queries (see Figure 5b), but a top- $k$  query itself also becomes more expensive. Thus in Figure 5a, 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 6, we compare the performance of proposed algorithms as we vary query interval length  $|I|$ . In terms of efficiency, Figure 6a shows T-Hop/S-Hop/S-Band is much faster than baseline solutions T-Base and S-Base. On the other hand, we also find that our proposed algorithms scale better with  $|I|$  than with  $k$  (recall Figure 5). The reason is that the time complexities of

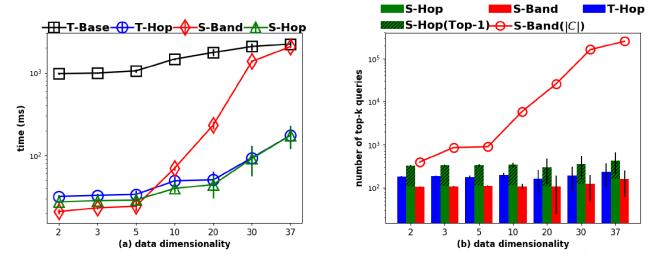
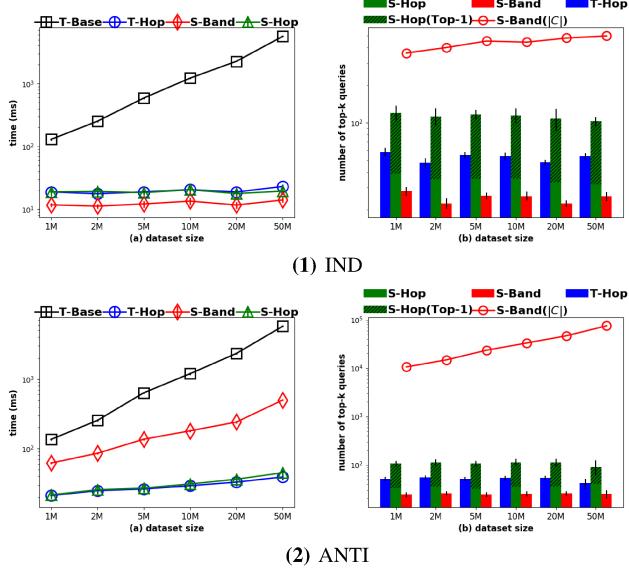


Fig. 7: Performance comparison on Network-X as  $d$  varies.

T-Hop/S-Hop and S-Band are quadratic in  $k$  but only linear on  $|I|$  (recall Lemma 3 and Lemma 4). In terms of number of top- $k$  queries, in Figure 6b, 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  $\tau$  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 7. Let us first take a look on Figure 7b. 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 7a. The query time of T-Base, T-Hop and S-Hop slowly increases as dimensionality increases, because top- $k$  queries in 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 ( $< 5$ ) data, in higher dimensions 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 8 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 8-1, we can see that T-Hop, S-Hop and S-Band scale well on large IND datasets, and S-Band again performs slightly 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 8-1b 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 7. Overall, on IND data,  $|C|$  is only about 4-5



**Fig. 8:** Scalability test on IND and ANTI Syn-X.

**TABLE IV:** Query time (in seconds) comparison on NBA-2 when varying  $\tau$ . PostgreSQL backend.

$\tau$ (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

times bigger than the actual answer size, which will not incur a big sorting overhead for S-Band. However, the situation is different for ANTI Syn-X. in terms of query efficiency, T-Hop and S-Hop still scale well, but S-Band becomes much more expensive. 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, 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.

### B. 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 [25] 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

**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

**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

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.<sup>7</sup> 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 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  $\tau$  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  $\tau$  (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  $\tau$ ). 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.

### C. 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  $\tau$ , 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

<sup>7</sup>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.

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 [26] and [27], 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 [28] 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 [29]–[31] and [32]. 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. [32] 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 [33] and [34], 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.

**Acknowledgements.** This work was supported by NSF grants IIS-1718398, IIS-1814493, CCF-2007556, and a grant from the Knight Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

## 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. Matousek, “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] J. Gao *et al.*, “Durable top- $k$  instant-stamped temporal records with user-specified scoring functions,” Duke Univ., Tech. Rep., 2020, <https://arxiv.org/abs/2102.12072>.
- [15] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf, “Computational geometry,” in *Computational geometry*, 1997.
- [16] 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.
- [17] P. K. Agarwal *et al.*, “Range-max queries on uncertain data,” *Journal of Computer and System Sciences*, vol. 94, 2018.
- [18] 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.
- [19] 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.
- [20] A. Mehta, A. Saberi, U. Vazirani, and V. Vazirani, “Adwords and generalized on-line matching,” in *FOCS*, 2005.
- [21] P. K. Agarwal, H. Kaplan, and M. Sharir, “Union of hypercubes and 3d minkowski sums with random sizes,” in *ICALP*, 2018.
- [22] 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.
- [23] S. Har-Peled and B. Raichel, “On the complexity of randomly weighted multiplicative voronoi diagrams,” *Discrete & Computational Geometry*, vol. 53, 2015.
- [24] *Codes Repository*, <https://anonymous.4open.science/r/57bd8e0e-9ba2-4e21-95c8-3750007e708e>.
- [25] *PostgreSQL*, 2019, <https://www.postgresql.org/>.
- [26] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu, “Prominent streak discovery in sequence data,” in *SIGKDD*, 2011.
- [27] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li, “Discovering general prominent streaks in sequence data,” *TKDD*, vol. 8, 2014.
- [28] B. Jiang and J. Pei, “Online interval skyline queries on time series,” in *ICDE*, 2009.
- [29] M. L. Lee, W. Hsu, L. Li, and W. H. Tok, “Consistent top- $k$  queries over time,” in *DASFAA*, 2009.
- [30] H. Wang, Y. Cai, Y. Yang, S. Zhang, and N. Mamoulis, “Durable queries over historical time series,” *TKDE*, vol. 26, 2014.
- [31] J. Gao, P. K. Agarwal, and J. Yang, “Durable top- $k$  queries on temporal data,” *VLDB*, vol. 11, 2018.
- [32] N. Mamoulis, K. Berberich, S. Bedathur *et al.*, “Durable top- $k$  search in document archives,” in *SIGMOD*, 2010.
- [33] K. Semertzidis and E. Pitoura, “Durable graph pattern queries on historical graphs,” in *ICDE*, 2016.
- [34] K. Semertzidis *et al.*, “Top- $k$  durable graph pattern queries on temporal graphs,” *TKDE*, vol. 31, 2018.