Durable Queries over Historical Time Series

Hao Wang, Yilun Cai, Yin Yang, Shiming Zhang, and Nikos Mamoulis

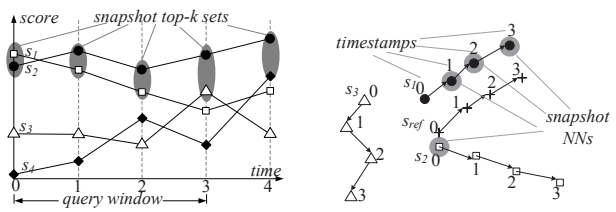
Abstract—This paper studies the problem of finding objects with durable quality over time in historical time series databases. For example, a sociologist may be interested in the top 10 web search terms during the period of some historical events; the police may seek for vehicles that move close to a suspect 70% of the time during a certain time, etc. Durable top-k (DTop-k) and nearest neighbor (DkNN) queries can be viewed as natural extensions of the standard snapshot top-k and NN queries to timestamped sequences of values or locations. Although their snapshot counterparts have been studied extensively, to our knowledge, there is little prior work that addresses this new class of durable queries. Existing methods for DTop-k processing either apply trivial solutions, or rely on domain-specific properties. Motivated by this, we propose efficient and scalable algorithms for the DTop-k and DkNN queries, based on novel indexing and query evaluation techniques. Our experiments show that the proposed algorithms outperform previous and baseline solutions by a wide margin.

Index Terms—Durable query, time series, historical data, spatio-temporal databases.

1 Introduction

The top-k query [11], which selects k best objects based on their ranking scores, is a common approach to obtaining a small set of desirable objects from a large database. Recently, top-k search has been extended to databases that contain multiple versions of data objects, e.g., web archives, trajectory data, time series, etc. Ranked retrieval in such applications may need to consider not only an object's value at one particular time instance, but also its overall quality during a time period [9][15][20].

In this paper, we study in depth the problem of finding objects of consistent quality during a time interval. We first study the *durable top-k (DTop-k) query* that operates on a historical database where each object is a 1D time series, i.e., at each time instance, every series carries a single scalar. Given k, time interval $[t_b, t_e)$ (called the *query window*), and percentage $0 < r \leq 1$ (called the *durability threshold*), a DTop-k query retrieves objects that appear in the snapshot top-k sets for at least $\lceil r \cdot (t_e - t_b) \rceil$ timestamps during $[t_b, t_e)$. Figure 1(a) shows an example with 4 series s_1–s_4. Assuming higher scores are preferred, a durable top-2 query with $[t_b, t_e) = [0, 4)$, $r = 70\%$ retrieves s_1 and s_2, since they appear in the top-2 set in at least 70% timestamps during $[0, 4)$.



(a) Example 1D DTop-k query (b) Example 2D DkNN query

Fig. 1. Example durable queries

We also identify a natural extension of the DTop-k query: the durable k nearest neighbor (DkNN) query, which considers at each time moment the k nearest neighbors of a reference series s_{ref}. Consider again the example of Figure 1(a), and the DkNN query with $s_{\text{ref}} = s_4$, $k = 1$, $[t_b, t_e) = [0, 4), r = 70\%$; i.e., we are interested in the sequences that are the nearest neighbor of s_4 on at least 70% of the timestamps 0-3. The only series that qualifies this query is s_3, since it is the NN of s_4 75% of the time in $[0, 4)$. As we show in the paper, the DkNN query is much more challenging compared to DTop-k, since the former is rather resistant to materialization and indexing.

Durable queries are useful in many real world applications. For example, consider Google Zeitgeist[1], which presents weekly statistics of search keywords, each of which is associated with a time series of its search volumes. A DTop-k (resp. DkNN) query can be used to identify keywords that are frequently searched (resp. most related) during some time period, which may be further used by sociologists to understand the impact of certain historical events. A similar application is Twitter Trendsmap[2], which tracks frequently mentioned phrases and hashtags. In SciScope[3], a geospatial search engine built upon a wide-area sensor network, durable queries may be used by meteorologists to identify regions with consistently high environmental indices in particular time windows. In general, durable queries may serve as fundamental tools in time series analysis; domain experts can use their results to better understand their data and trigger further investigation. We demonstrate some interesting examples in Section 6.

Durable queries can naturally be extended for *multi-*

1. http://www.google.com/intl/en/press/zeitgeist2010/
2. http://trendsmap.com/
3. http://www.sciscope.org/

Digital Object Indentifier 10.1109/TKDE.2013.10

dimensional time series, where at each time moment every series carries an array of values. For top-k search, in order to rank the series at every time instance, we need to define an aggregate scoring function on the values in the individual dimensions (e.g., a linear function). For NN queries, we use a distance measure (e.g., Euclidean distance) at each timestamp between the reference series s_{ref} and the sequences; for example, a police officer may investigate on vehicles consistently moving close to a pivot, e.g., a suspect or a witness. Figure 1(b) illustrates an example 2D time series dataset containing the positions of three moving objects s_1-s_3 at timestamps 0-3. Considering a DkNN query with $k = 1$ and a period $[0, 4)$, object s_1 satisfies the query for $r \leq 75\%$, since it is the snapshot NN of s_{ref} for timestamps 1-3.

To our knowledge, currently there is a very narrow selection of solutions for the DTop-k query, and no previous work on the DkNN query. The only existing solutions for DTop-k (reviewed in Section 2) employ either brute-force search, or techniques that are limited to specific domains. To fill this gap, we propose an efficient method called *top-k event scanning* (TES). TES exploits the fact that real-world time series typically exhibit a certain degree of smoothness, meaning that the changes in the top-k set at adjacent timestamps are usually small, if at all. TES indexes these changes and incrementally computes the snapshot top-k sets at each timestamp of the query window. To efficiently support DkNN queries on 1D time series, we extend the methodology of TES, and propose an efficient solution, *query space indexing* (QSI), that indexes the query space. Going one step further, we extend QSI to handle multi-dimensional top-k and k-NN queries. Extensive experiments using real and synthetic data confirm that the proposed methods significantly outperform previous ones, often by large margins. In the following, Section 2 surveys related work; Sections 3 and 4 describe the proposed solutions for DTop-k and DkNN queries on 1D time series, respectively; Section 5 discusses query processing on multi-dimensional series; Section 6 evaluates the proposed methods experimentally; Section 7 concludes the paper.

2 Related Work

2.1 Consistent and Durable Queries

Lee et al. [15] were the first to study the consistent top-k query, which is the special case of DTop-k with the durability threshold r fixed to 100%. In the example of Figure 1(a), a consistent top-2 query with time period $[0, 5)$ retrieves only object s_2. The basic idea of the solution in [15] (referred to as LHL) is to exhaustively verify every object in the dataset against the query definition. For each object s, LHL first checks whether s belongs to the top-k set at timestamp t_b. If so, LHL continues to check if s is a top-k object at $t_b + 1$; otherwise, it discards s and starts with another object. The process continues, until either s is eliminated, or after checking the rank of s at every timestamp in the query window. In order to accelerate snapshot top-k membership checking, LHL pre-computes the rank of each object at every timestamp, and organizes this information into a sorted list, stored on disk in a compressed format. For instance, in Figure 1(a), LHL associates the list $(1, 2, 2, 3, 3)$ to object s_1, signifying that s_1 ranks 1st at timestamp 0, 2nd at time 1-2, and 3rd at time 3-4. During query processing, LHL scans the rank list of an object linearly from t_b, until reaching either t_e or a value larger than k.

LHL does not support DTop-k queries with $r < 100\%$. To handle such cases, we extend LHL as follows. For each object s, we scan the part of its rank list from timestamp t_b to t_e, and count the number of times that s is in the snapshot top-k sets. During the scan, if we find that s is outside the top-k set for more than $(1 - r) \cdot (t_e - t_b)$ timestamps, we drop s since it cannot possibly reach the durability threshold r. The set of objects that pass the verification are reported as results. The main drawback of LHL is that it scales poorly with the number of objects, as each object initiates a list scan with at least one I/O read.

U et al. [20] studied durable top-k queries in the context of keyword search in web archives, where each object is a web document that gets edited or replaced over time. In addition to the parameters k, $[t_b, t_e)$, and r, a durable query in [20] also involves a keyword list K_w. The score of a document version is calculated based on its relevance to the keywords in K_w with an IR model. There are important differences between our work and [20]. First, computing the relevance of a document to an arbitrary K_w is both hard and expensive. Therefore, preprocessing methods cannot be used to accelerate search, as in Ref. [15] and in our work. Second, the data domain, i.e., versional documents, is quite special: the relevance of keywords to documents remains relatively constant in adjacent timestamps. When this assumption does not hold, e.g., if all objects change values at every timestamp, as in Figure 1(a), the methods in [20] reduce to brute-force search. Hence, the solutions in [20] are tailored to a specific domain, and are not suitable for DTop-k queries in the general case.

2.2 Other Related Temporal Queries

Numerous solutions (e.g., [1][8][14]) have been proposed for indexing time series to support similarity search. Such queries retrieve time series that are closest to a reference series, according to a certain distance measure. Two popular distance measures are (i) the Euclidean distance in the space defined by considering each time instance as a dimension and (ii) dynamic time warping (DTW) [14], which improves robustness over the Euclidean distance by allowing

mapping of shifted sequence elements. The Gemini framework [8] addresses the dimensionality curse in time-series indexing and search using dimensionality reduction; popular methods in this direction include Chebyshev polynomials [2], piecewise linear approximation [5], APCA [3], etc. These methods do not apply to durable queries, as they focus on an object's overall similarity to a query, rather than their properties at individual timestamps.

There is also a vast amount of existing work in indexing and searching trajectories, where each record contains the locations of a moving object at different timestamps. TrajStore [6] is a full-featured storage engine for trajectories using adaptive quad-tree indices. Sherkat and Rafiei [18] propose a new class of robust summaries for high-dimensional time series. Chen et al. [4] propose a new distance measure for multi-dimensional time series, as well as the corresponding indexing and searching methods. Another line of work focuses on spatio-temporal queries on moving objects trajectories. Yu et al. [22] propose efficient methods for continuous nearest neighbor search, which continuously updates the nearest neighbor of a query as objects update their locations. Güting et al. [9] study a similar problem, termed TCkNN, but focus on retrieving the nearest neighbors during a historical period rather than the current ones. Specifically, a TCkNN query finds, at each timestamp during the given period, the NN of a reference trajectory. The technical focus in [9] is to organize trajectories into an R-tree-like structure [10], and then take advantage of some pruning heuristics.

Compared to similarity queries, there is little work on top-k queries for time series data, despite the importance of such queries. Although it is possible to simulate a top-k query by a k-NN query with an imaginary reference time series that has the largest domain value at each time moment, such a reduction is often "far from satisfactory" [16]; methods designed for similarity search do not capture well the unique properties of top-k search. Li et al. [16] conducted a thorough study on the evaluation of snapshot top-k queries (i.e., find the top-k objects at a given timestamp) on continuous time series with a piecewise linear representation. The focus of [16] is clearly different from ours, both in terms of the query nature and the data model used.

Jestes et al. [12] study *aggregate top-k queries* on temporal data with a piecewise linear representation. The goal is to find the top-k objects with the highest aggregation scores (e.g., average, sum, etc.) in a given time interval. The focus and the data model of [12] are clearly different from ours, and their solutions do not apply to durable queries. Another piece of related work is the interval skyline query [13]. An object s_i dominates another s_j, if an only if s_i is better than s_j in at least one timestamp, and no worse in all other timestamps. The set of objects that are not dominated is then reported as the interval skyline. The interval skyline and the durable top-k, however, retrieve very different results. The former's result set includes objects with high values in a small number of timestamps, whereas the latter identifies objects with durable quality. For instance, in Figure 1(a), s_1 is on the interval skyline as long as the query window contains timestamp 0 (where s_1 is the best object), regardless of its scores in other time instances. Consequently, the solutions of [13] are inapplicable to our problems. The probabilistic top-k query [17], which finds objects with high probability to be in the top-k set, is also remotely related to this work, since one can view each timestamp as a possible world, and calculate the probability for each object. On the other hand, the focus of [17] is clearly different from ours, and their methods do not apply to durable queries.

Finally, *recent-biased* time series (i.e., recent timestamps are assigned higher weights than older ones) have been studied in the context of online analysis of streaming data. The focus of this work is different, however, since we focus on *offline* queries over *historical* data. For instance, in the various application scenarios mentioned in Section 1, it is generally more natural to consider timestamps within the query window as equally important, than giving higher weights to more recent time instances. For this reason, in the following we focus on the equal-weight time series model, as is done in many existing work involving historical data, e.g., [12][13][15]. Possible extensions of the proposed algorithms to handle recent-based time series is discussed in Section 7.

3 Durable Top-k Processing

This section focuses on DTop-k processing in settings where each object s is associated with a single value (i.e., its score) at each timestamp t. In other words, the top-k scores of the objects at all timestamps are known before query time. In practice, the value of k is usually only a fraction of the total number of objects in the database [11]. Hence, we use $k_{\max}$ to denote the largest supported value of k in the target application. For the ease of presentation, we assume that all series in the dataset are sufficiently long to cover the query window, i.e., each of them has a value at every timestamp during $[t_b, t_e)$. If a series starts after t_b or terminates before t_e, we simply put a (conceptual) value of $-\infty$ on each of its undefined timestamps. Meanwhile, we use $\Delta_{\min} = \lceil r \cdot (t_e - t_b) \rceil$ to denote the minimum number of timestamps for which an object should satisfy the corresponding snapshot top-k query in order to appear in the DTop-k results.

Besides LHL [15] described in Section 2.1, another naïve solution (referred to as NAI) for the DTop-k query is to compute the snapshot top-k results at every timestamp, and report the objects that appear in no less than $\Delta_{\min}$ snapshot top-k sets. Clearly, this

technique has a high cost when the query window is long. In the following we present a novel solution TES that significantly outperforms both LHL and NAI. Section 3.1 describes the general framework of TES. Sections 3.2 and 3.3 present methods for storing and retrieving the differences of the top-k sets along a time interval. Section 3.4 discusses the general applicability of TES to any query that aggregates top-k results from multiple consecutive timestamps. Table 1 summarizes common symbols used throughout this section.

TABLE 1
List of Frequent Notations

Symbol	Meaning
$[t_b, t_e), W$	DTop-k query window and its length
$k, k_{\max}$	DTop-k query parameter and its max value
$r, \Delta_{\min}$	Minimum percentage and number of timestamps that a DTop-k result should stay in the snapshot top-k sets
N, T	Total number of objects and timestamps
S_t^*	Snapshot top-k set at timestamp t

3.1 General Framework of TES

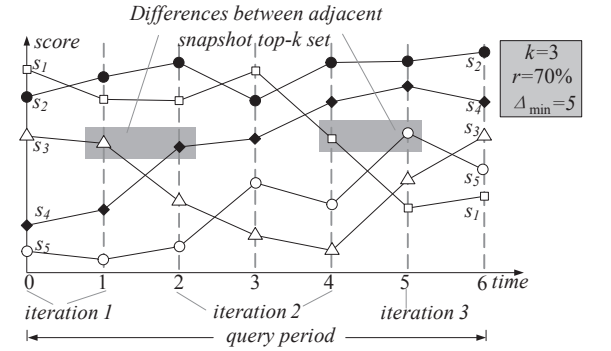
In many applications, due to time series continuity, the relative ranks of an object at consecutive timestamps tend to be stable. Accordingly, the top-k set may not change at every time instance; when it indeed changes, the differences between the old and new top-k sets are usually small. For instance, in Figure 1(a), the top-2 set remains the same at the first three timestamps, and changes only partially later. Thus, it is unnecessary to compute the snapshot top-k results from scratch at each timestamp. TES builds on this observation; Algorithm 1 shows its general framework. The basic idea of TES is to (i) compute the top-k set at timestamp t_b; (ii) find the next timestamp $t' > t_b$ where the top-k set changes and update it; (iii) repeat step (ii) until $t' > t_e$.

While the top-k set is being updated, the *durability* of the objects found in the set are also updated and the objects that make it to the top-k durable result are output. The durability Δ_s of an object s is defined by the number of timestamps in $[t_b, t_e)$ for which the object is in the top-k. Two sets of objects are maintained during the algorithm; a set CS of *candidate* objects that are not yet confirmed to be durable top-k results and a set RS of confirmed results. Whenever an object s is found for the first time in the top-k set, s is moved to CS if it is possible for s to make it to the top-k result, based on the number of remaining timestamps until t_e (line 7). As soon as a candidate object is found at least $\Delta_{\min}$ times in the top-k set, it is moved to RS (line 10). If there are not enough timestamps for new objects to make it in the result and CS is empty, the algorithm terminates, before having to reach t_e (line 13).

Example. Consider the data in Figure 2 and a DTop-3 query q with $[t_b, t_e) = [0, 7)$, and durability threshold $r = 70\%$, meaning that a result object must be in the top-k set for at least $\Delta_{\min} = 5$ timestamps. The steps of the algorithm are illustrated at the bottom of the figure. Initially, TES computes the top-k set $S_{t_b}^*$ at time $t_b = 0$, adds $\{s_1, s_2, s_3\}$ to CS, and sets $t = t_b$. The next timestamp $\geq t$ that the snapshot top-k set changes is $t' = 2$. Thus, TES increases the durability counters of all objects in the previous top-k set (i.e., $\{s_1, s_2, s_3\}$) by $t' - t = 2$. It also updates t to 2 and the top-k set to $S_t^* = \{s_1, s_2, s_4\}$. In the next iteration, $t' = 5$ is found and the durabilities of $S_t^* = \{s_1, s_2, s_4\}$ are increased by $t' - t = 3$. Since counters Δ_1 and Δ_2 reach $\Delta_{\min} = 5$, s_1 and s_2 are confirmed results, and moved to RS. Meanwhile, candidate s_3 is purged from CS, since it cannot meet the 5 timestamps durability threshold, even if it is a top-k object in all remaining time instances (i.e., 5-6). In the third iteration, the only candidate s_4 is promoted to RS, leaving CS empty. Now, there are only two timestamps left, insufficient for any new top-k object (e.g., s_5) to reach the durability threshold $\Delta_{\min} = 5$. Hence, TES terminates reporting $\{s_1, s_2, s_4\}$.

Algorithm 1 Top-k Event Scanning (TES)

```
   TES(q)
   // Input: q = {k, [t_b, t_e), r} is the durable top-k query
1: Δ_min ← ⌈r · (t_e − t_b)⌉, t ← t_b, CS ← ∅, RS ← ∅
2: while t < t_e do
3:     Retrieve from disk the snapshot top-k set S*_t at time t
4:     Find the next timestamp t' (t < t' < t_e) where S*_t' ≠ S*_t;
       if no such t' exists, t' ← t_e
5:     for each object s ∈ S*_t do
6:         if s ∉ CS ∪ RS and t ≤ t_e − Δ_min then
7:             Add s to CS with Δ_s = t' − t
8:         else if s ∈ CS then add t' − t to Δ_s
9:         if Δ_s ≥ Δ_min then
10:            Delete s from CS and add s to RS
11:    if t > t_e − Δ_min then
12:        Remove from CS every object s satisfying
                       Δ_s < Δ_min − t_e + t'
13:        if CS = ∅ then break
14:    t ← t'
15: return RS
```



Iteration	t	t'	CS(Object/Counter)	RS
1	0	2	$s_1/2$, $s_2/2$, $s_3/2$	-
2	2	5	$s_4/3$	s_1, s_2
3	5	6	-	s_1, s_2, s_4

Fig. 2. Example of TES

In the above example, TES only computes the top-k result at two timestamps (2 and 5) at which the

snapshot top-k set changes, and applies two updates to the top-k set (s_4 replacing s_3 at time 2 and s_5 replacing s_4 at time 5). Furthermore, the algorithm stops early, before reaching t_e.

Clearly, the most expensive modules in the TES framework are computing the snapshot top-k sets and finding the next timestamp. Since the data are historical, the overall query cost can be alleviated by (i) precomputing the snapshot top-k sets at every timestamp and (ii) indexing the timestamps where the top-k set changes. Compared to a naïve algorithm that re-computes the top-k results at each timestamp, TES clearly performs considerably fewer operations. The overall efficiency of TES, however, depends upon the implementation of the module that finds the next change of the top-k set. In the next sections, we discuss appropriate implementations for this module.

3.2 TES$_\delta$

We first describe an intuitive implementation of TES, namely TES$_\delta$. The main idea is, *for all possible values of k*, to directly materialize the changes between consecutive top-k sets. For instance, in Figure 2, changes to the top-3 set are: (i) s_4 replaces s_3 at timestamp 2, (ii) s_5 replaces s_1 at time 5 and (iii) s_3 replaces s_5 at time 6. Assuming that $k_{\max} = 4$, Table 2 lists, for all $k \leq k_{\max}$ the changes occurring between adjacent snapshot top-k sets; for example, $2(+s_4, -s_3)$ means that at timestamp 2, s_4 enters the top-3 set, and s_3 leaves. Although in this particular example, at each timestamp, at most one object enters or leaves any top-k set, multiple changes may happen to a top-k set in the general case.

TABLE 2
Changes in Snapshot Top-k Sets in Fig. 2

k	Timestamps (Changes)
1	1($+s_2$, $-s_1$), 3($+s_1$, $-s_2$), 4($+s_2$, $-s_1$)
2	4($+s_4$, $-s_1$)
3	2($+s_4$, $-s_3$), 5($+s_5$, $-s_1$), 6($+s_3$, $-s_5$)
4	3($+s_5$, $-s_3$), 5($+s_3$, $-s_1$)

TES$_\delta$ materializes the entire table that encodes the changes for each k along the timeline (e.g., Table 2); each row of the table is packed into disk blocks, stored in a separate file, and indexed with a B$^+$-tree with time as the key. Given a DTop-k query, the top-k set for $t = t_b$ is first retrieved from disk. Then, TES$_\delta$ searches the B$^+$-tree corresponding to the k-value of the query and finds the entry with the smallest timestamp $t' > t$. All timestamps between t and t' are skipped, and the new top-k set $S^*_{t'}$ at t' is computed by updating the previous top-k result S^*_t with the retrieved changes. For example, for a query with $k = 3$, $t_b = 0$, and $t_e = 6$, after the top-3 set $\{s_1, s_2, s_3\}$ is found at $t_b = 0$, the B$^+$-tree of the 3rd row is searched for the first entry with timestamp > 0; that is entry 2($+s_4$, $-s_3$). This entry implies that the top-3 set becomes $\{s_1, s_2, s_4\}$ at $t' = 2$. By linearly scanning the entries while $t' < t_e$, TES$_\delta$ retrieves all changes in the top-k set during the query interval.

Performance Analysis. The efficiency of TES$_\delta$ depends on the volatility of the dataset. Let $\Sigma \leq k \cdot (t_e - t_b)$ be the total number of times that any object enters or exits the top-k set during the query window, the total I/O cost of TES$_\delta$ is $\mathcal{O}\left(\log_B T + \frac{\Sigma}{B}\right)$, where B is the size of a disk block. In the worst case, the entire top-k set changes at every timestamp and TES$_\delta$ reduces to the naïve algorithm. Such cases are rare; TES$_\delta$ usually answers a DTop-k query with a significantly lower cost. The main drawback of TES$_\delta$, however, is that it imposes high storage overhead, because rank changes of objects are replicated across multiple rows of the table. In our example, the fact that s_1 moves from rank 1 to rank 3 at timestamp 4 is reflected in two entries timestamped 4: one in the first row and one in the second row, because s_1 exits the top-1 and top-2 sets at the same time. Due to this replication, in the worst case, the space complexity of TES$_\delta$ reaches $\mathcal{O}\left(k^2_{\max} \cdot T/B\right)$. We address this issue in a more sophisticated implementation of TES, presented next.

3.3 TES$_\lambda$

TES$_\lambda$ aims at achieving similar query performance as TES$_\delta$ with much lower space requirements. The main idea is to compress the changes in the snapshot top-k sets using a novel interval representation for rank changes. As discussed above, at any timestamp t, objects may enter or exit multiple top-k sets for different values of k; these k values form a continuous range, which we call a *rank-change interval*. In Figure 2, at timestamp 4, s_1 leaves both the top-1 and top-2 sets. Instead of replicating these changes to two rows (as done by TES$_\delta$ in Table 2), TES$_\lambda$ represents this event with a tuple $\langle -s_1, [1, 3)\rangle$, which signifies that s_1 leaves all top-k sets for $k \in [1, 3)$. Such a representation saves significant space for long rank-change intervals.

Figure 3 (left side) illustrates a more complex example involving 8 objects s_1–s_8, and 3 timestamps 0–2, with $k_{\max} = 8$; different markers (e.g., triangles, squares, etc.) are used for different objects. At timestamp 1, object s_4 leaves the top-4, top-5, and top-6 sets, generating a $\langle -s_4, [4, 7)\rangle$ rank-change interval. At the same time, s_8 generates a $\langle +s_8, [6, 8)\rangle$ event. The right side of Figure 3 presents a tree structure of 7 nodes storing all rank change events at timestamps 1 and 2, as 14 rank-change intervals. The directions of the arrows on the intervals indicate whether the object enters (up) or leaves (down) the corresponding top-k sets. For example, the rightmost interval indicates that s_1 leaves the top-3 and top-4 sets at time 2.

To answer a DTop-k query, TES$_\lambda$ retrieves all rank-change intervals that overlap with k during the query window, ordered by time. Then, from these intervals, TES$_\lambda$ extracts the corresponding objects, and applies

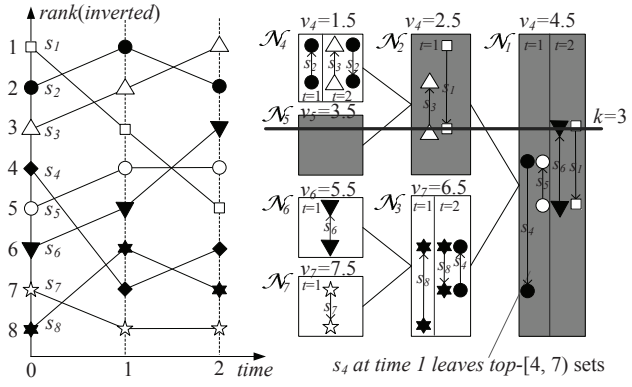


Fig. 3. Example CJI-Tree of 8 time series over 3 timestamps; 14 rank-change intervals are distributed into 7 tree nodes.

the changes to the previous top-k set. For instance, consider a DTop-3 query spanning the time window of Figure 3. No interval corresponding to timestamp 1 intersects with the horizontal line $k = 3$, indicating no change to the top-3 set at time 1; concerning the second timestamp, the two rightmost intervals overlap with $k = 3$, signifying that object s_1 leaves and s_6 enters the top-3 set.

Next we clarify how TES$_\lambda$ organizes and retrieves rank-change intervals. The classic data structure for intervals, which answers stabbing queries efficiently, is the interval tree [7]. A straightforward implementation of TES$_\lambda$ would be to construct an interval tree for each timestamp t, in order to efficiently obtain intervals containing k at each timestamp $t \in (t_b, t_e)$. However, this would require one tree search for each $t \in (t_b, t_e)$, i.e., high I/O cost for long query windows. Instead, TES$_\lambda$ employs a novel data structure, called the *conceptual joint interval tree* (*CJI-tree*). In Figure 3, the 14 intervals are organized into a CJI-tree with 7 nodes $\mathcal{N}_1$–$\mathcal{N}_7$. Essentially, the CJI-tree consists of 7 lists of intervals, one for each node.

The rank-change intervals are assigned to the CJI-tree nodes as follows. Each node $\mathcal{N}_i$ is associated with a value $\mathcal{N}_i.v$; these values define a hierarchical partitioning of the rank domain, explained soon. A rank-change interval I is added to the highest tree node $\mathcal{N}_i$ for which $\mathcal{N}_i.v \in I$. To insert I to the tree, I is first tested against the root node $\mathcal{N}_r$. If I contains $\mathcal{N}_r.v$, then I is stored at $\mathcal{N}_r$; otherwise, if I is to the left (resp. right) of $\mathcal{N}_r.v$, I is recursively tested against the root of the left (resp. right) subtree of $\mathcal{N}_r$. The values associated with the nodes must ensure that each interval overlaps with at least one of such values in the entire tree. The CJI-tree is a binary tree with exactly $\lceil \log_2 k_{\max} \rceil$ levels of nodes, among which the leaf nodes are associated with middle points between two adjacent ranks. In Figure 3, the leaves $\mathcal{N}_4$–$\mathcal{N}_7$ are assigned values 1.5, 3.5, 5.5 and 7.5, which are the mid-points of rank pairs (1, 2), (3, 4), (5, 6), and (7, 8). Values assigned to internal nodes are the averages of the values for their corresponding children. Continuing the example, for the next level $\mathcal{N}_2$ and $\mathcal{N}_3$, $\mathcal{N}_2.v = (\mathcal{N}_4.v + \mathcal{N}_5.v)/2 = 2.5$, and $\mathcal{N}_3.v = (\mathcal{N}_6.v + \mathcal{N}_7.v)/2 = 6.5$. Finally, for the root, $\mathcal{N}_1.r = (\mathcal{N}_2.v + \mathcal{N}_3.v)/2 = 4.5$. Since a rank-change interval has length at least one, it must intersect with at least one of the values in the CJI-tree.

The CJI-tree is stored on disk as follows. A separate file is created for each node $\mathcal{N}_i$ of the tree. The intervals of $\mathcal{N}_i$ are sorted and grouped by time (recall that each interval corresponds to the change in the ranking of an object at a specific timestamp). Each file is indexed by a B$^+$-tree using time as key.

Algorithm 2 shows how the top-k set at a given timestamp t is incrementally updated using the CJI-tree. The main idea is to search the tree for nodes that may contain intervals intersecting with rank k. For each such node $\mathcal{N}$, TES$_\lambda$ finds the partition corresponding to timestamp t, and then obtains the set of intervals overlapping k, as in the traditional interval tree search algorithm [7]. The top-k set changes corresponding to the retrieved intervals are applied to the current top-k set (lines 5-8). In our running example, to compute the top-3 set at timestamp 2, TES$_\lambda$ initializes the top-k set to the previous one $\{s_1, s_2, s_3\}$ at time 1, and searches the CJI-tree starting from the root N_1. It then scans the partition for $t = 2$, and retrieves two intervals overlapping k that correspond to objects s_1 and s_6. Thus, s_6 replaces s_1 in the current top-k set. After that, TES$_\lambda$ descends the tree to the left child $\mathcal{N}_2$ of $\mathcal{N}_1$. Since $\mathcal{N}_2$ does not contain a partition for $t = 2$, TES$_\lambda$ continues to $\mathcal{N}_5$, which is empty. Since $\mathcal{N}_5$ is a leaf, the algorithm stops the traversal, and returns the current top-k set $\{s_2, s_3, s_6\}$.

Note that the set of nodes (e.g., $\mathcal{N}_1$, $\mathcal{N}_2$, $\mathcal{N}_5$) corresponding to a specific k (e.g., $k = 3$) are fixed throughout DTop-k evaluation. This means that, to evaluate a DTop-k query, we first use the B$^+$-trees of the files corresponding to these nodes to find the smallest time point $> t_b$ in them, and then scan these linearly and concurrently from these time points.

Algorithm 2 Updating the Top-k Set using the CJI-tree

```
Compute_Top-k(k, t, S*_{t-1})
// Input: k is the query parameter; t is the current
          timestamp; S*_{t-1} is the snapshot top-k set at
          the previous timestamp t − 1
// Output: The snapshot top-k set S*_t at t
1: Initialize S*_t to S*_{t-1}
2: Use k to calculate the set of nodes, N, that needs to be accessed
3: for each node N ∈ N do
4:     Retrieve from disk the list of rank-change intervals residing
       at N that correspond to t, and overlap with k
5:     for each interval I retrieved in the last step do
6:         Let s be the object corresponding to I
7:         if s ∈ S*_{t-1} then remove s from S*_t
8:         else Add s to S*_t
9: return S*_t
```

Cost balancing using λ. Although TES$_\lambda$ saves stor-

age, it incurs additional I/O and CPU costs during query processing for traversing the CJI-tree. To strike a balance between storage and query response time, we introduce a parameter $\lambda(\geq 1)$ into TES$_\lambda$: we store in the CJI-tree only those rank-change intervals with length at least λ, since intuitively longer intervals lead to higher space savings. In particular, when $\lambda = 2^\ell$ with integer ℓ, the CJI-tree only needs $\lceil \log_2 k_{\max} \rceil - \ell$ levels to ensure that each interval is stored at one node. The rest of the rank change information are directly stored as in TES$_\delta$. To incrementally compute a snapshot top-k set, TES$_\lambda$ retrieves rank change information from both the TES$_\delta$ table and the CJI-tree. A smaller value of λ leads to lower storage cost but higher query overhead; the reverse is also true.

Performance Analysis. We first analyze the storage savings of TES$_\lambda$ with $\lambda = 1$. At each timestamp, in the worst case all top-k results are different from the previous timestamp, leading to $2k$ rank-change intervals. Since each interval is stored exactly once in the CJI-tree, the space overhead of the tree is bounded by $\mathcal{O}(k_{\max} \cdot T/B)$, where $k_{\max}$, T and B are the maximum supported value for k, total number of timestamps in the dataset, and block size, respectively. Note that this is significantly lower than the storage cost $\mathcal{O}\left(k_{\max}^2 \cdot T/B\right)$ of TES$_\delta$. When $\lambda > 1$, intervals with length up to λ are duplicated for up to λ times. Thus, the overall space complexity of TES$_\lambda$ is $\mathcal{O}(\lambda \cdot k_{\max} \cdot T/B)$.

Regarding query processing, the number of CJI-tree nodes that have to be searched and scanned is $\lceil \log_2 (k_{\max}/\lambda) \rceil$. These searches cost $\mathcal{O}(\log_2 (k_{\max}/\lambda) \cdot \log_B T)$ due to the use of B$^+$-trees. The data (i.e., intervals) that have to be retrieved from each node depend on the length of the query window $W = t_e - t_b$ and the number of objects that enter/exit the top-k set at each timestamp inside the window. Assuming that the total number of times any object enters or exits the top-k set during W is Σ, the scanning cost is $\mathcal{O}(\Sigma/B)$. Thus, the overall I/O cost for answering a DTop-k query is $\mathcal{O}(\lceil \log_2 (k_{\max}/\lambda) \rceil \cdot \log_B T + \Sigma/B)$, which degenerates to TES$_\delta$ when $\lambda \geq k_{\max}$.

3.4 Generality of TES

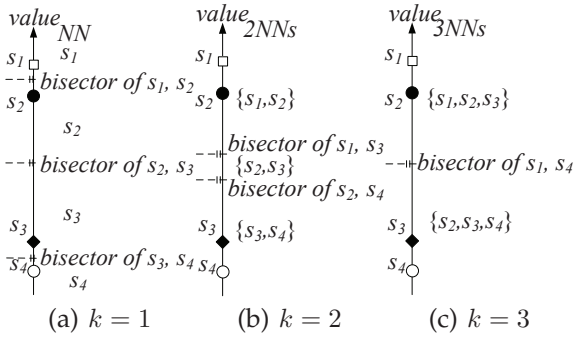
So far we have discussed the case where there is a given durability threshold r and only objects that pass this threshold are output. In some applications, it might be hard for the user to give an appropriate value for r. For such cases, an alternative is to ask the user to specify the number m of objects to be retrieves with the highest durability. TES can be easily adapted to support this version of the DTop-k query: instead of using the fixed threshold $\Delta_{\min}$ for pruning candidates, we use a *floating* $\Delta_{\min}$ threshold defined by the m-th durable object found so far, which we keep track of while updating the durabilities of the candidates.

In addition, although we have presented TES in the context of the DTop-k query, this method can be used to answer any query that post-processes *all* top-k query results inside a given time interval $[t_b, t_e)$. For example, a data analyst might be interested in the objects that enter and exit the top-k set the maximum number of times during the query window. In this case, the durability threshold can be replaced by an *instability* threshold.

4 Durable k-NN Processing

This section studies the evaluation of DkNN queries on 1D time series. Recall from Section 1 that a DkNN query contains a reference series s_{ref}, which has $t_e - t_b$ values, one for each timestamp in the query window. Hence, there is a vast space of possible DkNN queries, making effective materialization much more difficult. A naïve approach (referred to as NAI) is to simply compute the snapshot k-NN results at every timestamp, and combine them to answer the DkNN query. NAI is clearly inefficient, because (i) the k-NN set may not change at every timestamp and (ii) snapshot k-NN computations are expensive as they cannot be precomputed as in the DTop-k case. In the following we describe a novel solution, namely *query space indexing* (*QSI*), which indexes the results of all possible DkNN queries, and stores them compactly.

Figure 4 illustrates an example with 4 object values at a particular timestamp t. Assume that $k = 1$. Observe that as long as the value of the reference series s_{ref} at time t is above the bisector of s_1 and s_2, the snapshot NN of s_{ref} at t is always s_1. Meanwhile, s_1 cannot possibly be the NN of s_{ref}, when s_{ref} falls below the bisector of s_1 and s_2. Similarly, the snapshot NN of s_{ref} is s_2, if and only if $s_{\text{ref}}(t)$ lies between the bisector of s_1 and s_2, and that of s_2 and s_3. Accordingly, we split the value domain into 4 partitions as in Figure 4(a), using the bisectors of adjacent objects. The snapshot NN of s_{ref} at t can be derived based on the partition where $s_{\text{ref}}(t)$ lies in. Figures 4(b) and 4(c) show the situations for $k = 2$ and $k = 3$, respectively. Specifically, for $k = 2$, we partition the value domain with the bisector of s_1/s_3 and that of s_2/s_4; for $k = 3$, the split point is the bisector of s_1 and s_4.



(a) $k = 1$ (b) $k = 2$ (c) $k = 3$

Fig. 4. Example of value domain partitioning

In general, consider N time series $s_1, s_2, \cdots, s_N$ and a given time t. Without loss of generality, assume that $s_1(t) < s_2(t) < \cdots < s_N(t)$. Moreover, with respect to a given k, let $B_0 = -\infty$, $B_i = \frac{1}{2}(s_i(t) + s_{i+k}(t))$ for $i = 1, 2, \cdots, N-k$, and $B_{N-k+1} = +\infty$. Sequence $\{B_i\}$ defines a partition of the value domain $(-\infty, +\infty)$, $\Pi(t) = \{\pi_1, \pi_2, \cdots, \pi_{N-k+1}\}$, where $\pi_i = [B_{i-1}, B_i)$ $(i = 1, 2, \cdots, N-k+1)$. We have the following lemma.

Lemma 1: If the reference series s_{ref} satisfies $s_{\text{ref}}(t) \in \pi_i$, then the k-NN set of s_{ref} at time t is $S^*_t(\pi_i) = \{s_i, s_{i+1}, \cdots, s_{i+k-1}\}$.

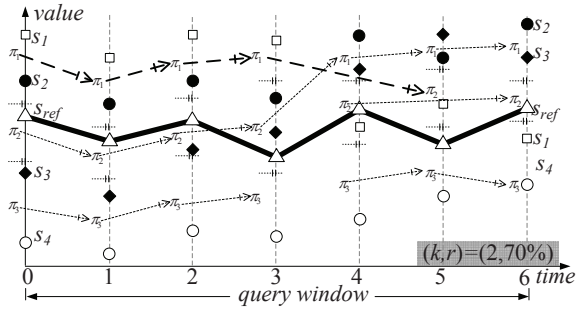
Proof: By definition it is clear that $s_{i-1}(t) < B_{i-1} < B_i < s_{i+k}(t)$. For any $s_{\text{ref}}(t) \in \pi_i$ and any j satisfying $i \le j \le i+k-1$, let d_j denote the distance $|s_{\text{ref}}(t) - s_j(t)|$, then we have $d_j < \min\{d_{i-1}, d_{i+k}\}$. Considering the fact that from $s_{i-1}(t)$ to $s_{i+k}(t)$ there are exactly $k+2$ distinct values, we thus achieve the conclusion: the k-NN set of s_{ref} at time t is indeed $S^*_t = \{s_i, s_{i+1}, \cdots, s_{i+k-1}\}$. □

Using Lemma 1, the k-NNs of a reference series s_{ref} at time t can be directly obtained from the corresponding set $S^*_t(\pi_i)$ of the interval π_i that contains $s_{\text{ref}}(t)$.

The proposed solution (QSI) materializes the above partitioning at every timestamp in the dataset; for every possible value of k, the corresponding partitions and their associated k-NN sets are stored in a file indexed by a B$^+$-tree with time as key. During the processing of a DkNN query, the snapshot k-NN set at any time t is directly *retrieved* from disk based on the value of $s_{\text{ref}}(t)$, rather than *computed* from scratch as in NAI. To further improve performance, QSI materializes additional information to avoid unnecessary k-NN retrievals. Specifically, observe that for each interval π_i at t (denoted as $\pi_i(t)$), its k-NNs can be identical to that of an interval π_j in the following timestamp $t'(> t)$, i.e., $S^*_t(\pi_i) = S^*_{t'}(\pi_j)$. QSI exploits this fact by linking such partition intervals together into a *partition-time index* (*PTI*), stored on disk along with the partitions. Figure 5 exhibits an example where the first interval π_1 at time 0 shares the same 2-NN set $\{s_1, s_2\}$ with the interval π_1 at timestamps 1 to 3, and the interval π_2 at 5. Accordingly, QSI links these intervals, starting from $\pi_1(0)$, as shown in the figure. Similarly, QSI links the interval $\pi_2(0)$ with $\pi_1(6)$ via $\pi_2(1), \pi_2(2), \pi_2(3), \pi_1(4)$ and $\pi_1(5)$, since they have the same 2-NN set $\{s_2, s_3\}$.

QSI performs the following steps to answer a DkNN query. First, it selects the interval π_* that contains the reference series s_{ref} at the first timestamp t_b of the query window, and returns the corresponding k-NNs. Then, the method skips the subsequent timestamps where the snapshot k-NN sets of s_{ref} can be derived from previous results, by following the links. The process continues, until QSI finishes processing all timestamps in $[t_b, t_e)$.

Figure 5 illustrates QSI on an example dataset, where $k = 2$, $[t_b, t_e) = [0, 7)$. The symbol '-' (e.g., at timestamps 2 and 3) indicates that the k-NNs of s_{ref} are the same as in the previous linked time instances. At $t_b = 0$, the interval containing s_{ref} is π_2, and its corresponding 2-NN set is $\{s_2, s_3\}$. $\pi_2(0)$ links to $\pi_2(1), \pi_2(2), \pi_2(3), \pi_1(4)$, $\pi_1(5)$ and $\pi_1(6)$, with the first three actually containing s_{ref}. Therefore, QSI skips timestamps 1 to 3 and starts a new iteration at timestamp 4. The same applies to the rest of the query window $[4, 7)$. The candidates after timestamp 4 are s_1, s_2, and s_3. s_3 is immediately reported as a final result since it already satisfies the durability requirement. s_1 is no longer a candidate since it cannot possibly reach the durability threshold. s_2 remains a candidate until being purged at timestamp 6.



Timestamp	0	1	2	3	4	5	6
$s_{\text{ref}} \in \pi_i$	π_2	π_2	π_2	π_2	π_2	π_3	π_2
2-NN of s_{ref}	s_2, s_3	-	-	-	s_1, s_3	s_1, s_4	-

Fig. 5. Example of QSI

Algorithm 3 summarizes the QSI algorithm. At each iteration, QSI checks the first timestamp t in the current query time queue Γ and picks the $\pi_i(t)$ which contains s_{ref}. Subsequent timestamps at which there is an interval containing s_{ref} and linked from $\pi_i(t)$ are collected into a set $\mathcal{T}$ (lines 4-6). QSI computes the k-NNs of s_{ref} at timestamp t, and removes $\mathcal{T}$ from the query time queue (line 7). If the number of the timestamps together with the current timestamp satisfies the durability condition $|\mathcal{T}| + 1 \ge \Delta_{\min}$, the k-NNs S^*_t are moved to the result set RS. Otherwise, all objects in S^*_t are merged into the candidate list CS and their counters are updated with $|\mathcal{T}| + 1$ (lines 8-11). Finally, all candidates are confirmed as final results (line 12) or purged as non-results (line 13). The iterations terminate when all query timestamps are processed or the candidate set becomes empty.

QSI accelerates DkNN processing in two ways. First, similar to TES, QSI can skip the retrieval/computation of snapshot k-NNs, when the reference series stays in the same linked intervals. Second, QSI operates on a compact representation of the data, which only keeps the rank list of the time series IDs, rather than their specific values at each timestamp. The links between intervals in different timestamps are kept into a separate index file. Hence, QSI is expected to be more efficient than NAI, especially when snapshot k-NNs change infrequently. The main drawback of QSI, however,

Algorithm 3 Algorithm QSI for DkNN queries

```
QSI(q)
// Input: q = {s_ref, k, [t_b, t_e), r} is the durable k-NN query
Δ_min ← ⌈r · (t_e − t_b)⌉, CS ← ∅, RS ← ∅
Initialize the priority query time set Γ ← [t_b, t_e)
while Γ ≠ ∅ and CS ≠ ∅ do
    t ← first(Γ) and remove t from Γ
    Retrieve from disk the partition π_i(t) of s_ref at timestamp t
    T ← {t' ∈ Γ | s_ref(t') ∈ π'_i(t') and π'_i(t') is derivatively
          linked by π_i(t)}
    Retrieve from disk the k-NN set S*_t for π_i(t) and Γ ← Γ − T
    if |T| + 1 ≥ Δ_min then
        RS ← RS ∪ S*_t, and CS ← CS − S*_t
    else
        Add each s ∈ S*_t into CS with Δ_s increased by |T| + 1
    Add {s ∈ CS | Δ_s ≥ Δ_min} into RS
    Remove {s ∈ CS | Δ_s + |Γ| < Δ_min} from CS
return RS
```

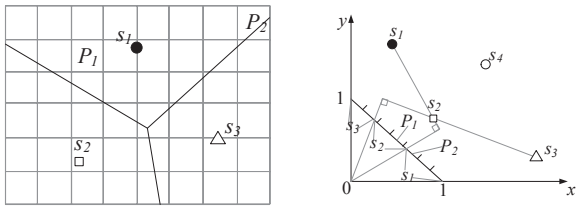
is its pre-computation cost for building the PTI for different values of k. Nevertheless, since k in k-NN queries are typically much smaller compared to top-k queries, we expect this cost to be bearable.

Performance Analysis. The efficiency of QSI depends on how well the query s_{ref} is consistent with the links. At every timestamp t, $O(\log_B(N-k))$ I/O is required to locate s_{ref} into a certain partition $\pi_i(t)$. Let $\alpha \leq t_e - t_b$ be the total number of times that s_{ref} deviates from the PTI links within the query window, then a cost $O\left(\frac{\alpha \cdot k}{B}\right)$ is needed to retrieve the k-NN sets. Therefore the overall query cost of QSI is $O\left((t_e - t_b) \cdot \log_B(N-k) + \frac{\alpha \cdot k}{B}\right)$. In the worst case $\alpha = t_e - t_b$, making QSI no better than the naïve solution that retrieves k-NN set at every timestamp; however such cases are rare. As to the storage cost, for each $k \leq k_{\max}$, $O((N-k) \cdot T/B)$ space is used to store the bisectors. Therefore the total storage cost is $\sum_{k=1}^{k_{\max}} O((N-k) \cdot T/B) = O(k_{\max} \cdot T \cdot (N - k_{\max} - 1)/B)$.

5 Multi-Dimensional Queries

QSI (described in Section 4) can be adapted to answer DkNN queries on multi-dimensional time series, e.g., trajectories. In this problem, every data object (and the reference series) has a multi-dimensional value at each timestamp, and the distance between two such values is given by the Euclidean metric. Given a timestamp t, a part of value space with identical k-NNs at t is an order-k Voronoi cell [7].

Figure 6(a) shows an example 2D durable k-NN query with three data objects s_1-s_3, and their corresponding Voronoi cells, assuming that $k = 1$. For instance, all possible positions of s_{ref} that fall into the upper cell containing s_1 have s_1 as their nearest neighbor. Similar to the 1D case, QSI partitions the value space using a regular grid, whose granularity Π is a user-defined parameter. Given a partition P and a timestamp t, the k-NNs of P at t can be uniquely determined, if P is completely contained in a single Voronoi cell, e.g., P_1 in the example. In this situation, the PTI index contains the snapshot k-NNs of P at time t, as well as the next timestamp $t' > t$ that k-NN set changes. Otherwise (e.g., partition P_2), QSI marks in the PTI index that the k-NN set of P as undetermined. The query processing algorithm is the identical to that of the 1D case (i.e., Algorithm 3).



(a) *QSI for DkNN* (b) *QSI for DTop-k*

Fig. 6. Example multi-dimensional durable queries

QSI can also be extended to answer multi-dimensional DTop-k queries where the scores of the objects are obtained by a linear function f as part of the query. Figure 6(b) illustrates a 2D example with 4 objects s_1-s_4. Suppose that all objects have positive values on both dimensions, and that the score function takes the form $f = ax + (1-a)y$; $0 \leq a \leq 1$. The space for all possible ranking functions can be represented by the line $x + y = 1$, on which each point p represents the score function that intersects $x + y = 1$ on p. Different portions of the line segment $x + y = 1$, $0 \leq x \leq 1$ correspond to score functions with different top-k results [21]. For instance, the parts of the line that correspond to s_1, s_2, s_3 as the top-1 result are shown in Figure 6(b). QSI splits the line segment $x + y = 1$ into a user-defined number Π of partitions, each of which may have a deterministic top-k set (e.g., P_1, whose top-1 is s_2), or not (e.g., P_2). The PTI index can be constructed accordingly, and the query processing module can directly be used. We found in our experiments that QSI is most effective on 1D DkNN queries. Its efficiency gains for 2D queries is relatively small; for higher (> 2) dimensional queries, the query cost reduction of QSI is marginal, and does not justify its high storage overhead. Hence, how to effectively handle high-dimensional durable queries remains an open problem.

6 Experimental Evaluation

We implemented all proposed algorithms and their competitors in C++. The experiments were run on a Linux 2.6.28 server with an Intel Core 2 Quad 2.66GHz CPU and 4GB of RAM. The page size is set to 4KB, the default page size of the OS. We use four datasets: (i) *AOL* (from www.gregsadetsky.com/aol-data) is a real web search log of around 650k users from Mar 1 to May 31, 2006. We count the frequency of every search term on each day, and obtain a time series dataset of 9098 terms over 92 days. Terms with very low frequencies are ignored. (ii) *Stock* (from wrds-web.wharton.upenn.edu) contains real daily closing

prices of 13k stocks traded in the New York Stock Exchange from Jan 1, 2000 to Dec 31, 2009 (2515 working days in total). We consider each stock as a time series, and each working day as a timestamp; stocks are ranked in decreasing order of their prices. (iii) *Air Pollution Index* (*API*) (from www.epd-asg.gov.hk) summarizes the hourly air pollution condition in various districts of Hong Kong, for the past 10 years. The total number of series and timestamps are 14 and 95844 respectively, and smaller pollution values are preferred in the top-k ranking. (iv) Synthetic datasets were generated with a random walk model [19], which contains 5k series and 10k timestamps. For each time series s_i, we first generate an initial value $s_i(0)$ uniformly from $[0, 100]$. Then, for each timestamp $t \in [1, 10000)$, we randomly choose a value $\alpha(t)$ from the normal distribution with mean 0 and standard deviation σ, and set $s_i(t)$ to $s_i(t-1) + \alpha(t)$. σ is a parameter evaluated in the experiments.

Before evaluating efficiency, we demonstrate the usefulness of durable queries with several sample queries on *AOL* and *Stock*. Table 3 shows the results of three DTop-100 queries over *AOL*, using $r = 50\%$. The first query has a time window of one month. As expected, most results are general terms related to daily lives (e.g., google, yahoo, map), which have relatively high durability. As we narrow down the time window to a fortnight (second query) or a week (third query), we start seeing results related to certain historical events. For clarity, results of the first query are not repeated for the second and the third query. For example, *American Idol*, a popular singing competition, was approaching a season finale during that time. Also, May 14 was the Mother's Day in 2006 and "mother" is a popular keyword in the third query.

TABLE 3
Sample DTop-100 query results on *AOL*

Query	Term (Durability)
1/5/2006 ≀ 31/5/2006	google (100%), free (100%), yahoo (100%), ⋯, home (96.8%), mapquest (96.8%), ⋯, sale (74.2%), university (71.0%), ⋯ unit (54.8%), john (54.8%)
3/5/2006 ≀ 18/5/2006	star (62.5%), medical (62.5%), mother (50%), idol (50%), love (50%)
6/5/2006 ≀ 14/5/2006	mother (88.9%), star (66.7%), love (55.6%), restaurant (55.6%)

Results of sample DTop-10 queries on *Stock* data are shown in Table 4. The table contains the 5 stocks that appear most frequently in the daily top-10 stocks by turnover during the specified query periods. Observe that the results of queries with different time windows can be radically different. These experiments indicate that durable queries can be very useful to financial time series analysts; further analysis of their results is beyond the scope of this paper.

Table 5 summarizes the query parameters used in the efficiency experiments. In each experiment, we vary one parameter and set all others to their defaults. Note that the API dataset uses smaller values for parameters k and $k_{\max}$, since it contains relatively few (14) time series. The parameter $w = (t_e - t_b)/T$ is the relative query window length, where T is the total number of timestamps. The start point of the query window t_b is randomly chosen from the range [0, $T - \lceil w \cdot T \rceil$], and t_e is calculated accordingly. For DkNN queries, the reference time series s_{ref} is computed by averaging 10 random data series at each timestamp.

TABLE 4
Sample DTop-10 queries and the results on *Stock*

Query	Stock (Durability)
1/3/2001 ≀ 31/3/2001	MICROSOFT (100%), CISCO (100%), NASDAQ (96.8%), INTEL (93.5%), ORACLE (85.5%)
3/1/2007 ≀ 31/12/2007	SPDR (100%), ISHARES (100%), APPLE (97.6%), GOOGLE (84.8%), EXXON MOBILE (68%)
1/7/2009 ≀ 31/8/2009	BOA(100%), SPDR(100%), ISHARES(95.2%), STREETTRACKS (85.7%), CITIGROUP (57.1%)

TABLE 5
Query parameters (default values in bold)

Parameters	Values	
k	*Stock, AOL*	1, 10, 25, 50, **100**, 250, 500
	API	1, 2, **4**, 6
$k_{\max}$	*Stock*	32, 64, 128, 256, **512**
	API	**8**
r	50%, 60%, **70%**, 80%, 90%	
$w = (t_e - t_b)/T$	1%, 5%, **10%**, 15%, 20%	

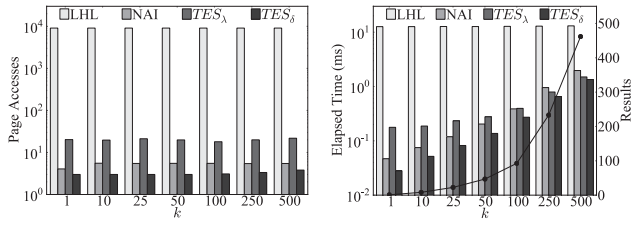
In all the experiments, we execute the methods 100 times, and average their number of page accesses and CPU time. It is worth mentioning that, although our datasets are small enough to fit into memory, we decide to implement all methods on disk-based data, and use cold buffers. The reason is that our methods are not dedicated to specific dataset sizes, and other real datasets (e.g., Google Zeitgeist data or an AOL-like web search log over a much longer time period) can be too large to fit into the memory. We used relatively small datasets, because (i) some of the (naïve) methods do not scale well with the database size and (ii) we could not find larger real datasets. Still, the results on these data provide useful conclusions about the relative efficiency of the methods. In the following, Sections 6.1 and 6.2 present the results for DTop-k and DkNN experiments, respectively.

6.1 Durable Top-k Evaluation

We first evaluate methods LHL [15], NAI, TES$_\delta$ and TES$_\lambda$ for DTop-k queries. To be fair, we compare with an efficient NAI implementation, which employs a file organization that materializes the top-$k_{\max}$ rankings for all timestamps, orders them by time and packs them to disk pages. This file is then indexed by a

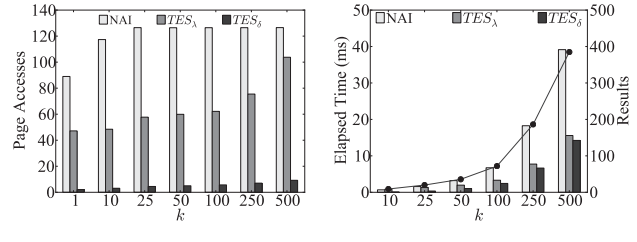
B$^+$-tree. Given a DTop-k query, NAI uses the B$^+$-tree to find the first top-$k_{\max}$ ranking inside the query time range and then sequentially accesses all top-k rankings in the file that are relevant to the query. Note that the worst-case query costs of LHL and NAI are $O\left(N \cdot \frac{t_e - t_b}{B}\right)$ and $O\left(\log_B T + (t_e - t_b) \cdot \frac{k}{B}\right)$ respectively, which are costly when N is large or the query window is long. In addition, for TES$_\lambda$, we fix the value of λ to 1, meaning all rank changes are stored in the CJI-tree, which maximizes space efficiency but increases query processing costs. The impact of λ is evaluated towards the end of this subsection.

Figures 7(a) and 7(b) plot the I/O cost and CPU time, respectively, as functions of parameter k on the *AOL* dataset, and Figures 8(a) and 8(b) on *Stock* data. Parameters w and r are set to their defaults. On *Stock*, we exclude the results for LHL [15], because its I/O and CPU costs are at least an order of magnitude higher than the remaining methods in all settings, as LHL exhaustively checks all 13k series. Clearly, TES$_\delta$ consistently beats both naïve methods in all settings, in terms of both I/O and CPU costs. While TES$_\lambda$ is not so competitive with very short queries (Figures 7(a) and 7(b), where the query window has length $\lceil 92 \times 10\% \rceil = 10$) ,it shows significant advantage when the query window gets longer (Figures 8(a) and 8(b), where the query length is $\lceil 2515 \times 10\% \rceil = 252$) .



(a) IO cost (b) CPU time

Fig. 7. Effect of k on *AOL*



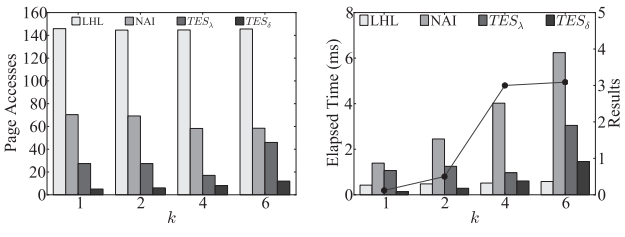
(a) IO cost (b) CPU time

Fig. 8. Effect of k on *Stock*

The I/O cost of NAI is not sensitive to parameter k, because it retrieves the materialized top-$k_{\max}$ results for each timestamp from disk during the query window. The I/O cost of TES increases with k, for two reasons. First, larger k leads to the retrieval of more rank change information (the probability that the top-k result changes increases with k) . Second, a larger k decreases the chance for early termination; manually checking reveals that when $k > 25$, early termination rarely occurs. Comparing the two variants of TES, TES$_\delta$ is considerably more efficient than TES$_\lambda$ in terms of I/O, since the latter requires accessing multiple nodes in the CJI-tree. The main advantage of TES$_\lambda$, however, is its flexibility, evaluated later.

The CPU time of all methods grows linearly with k. Both versions of TES consistently beat NAI by a wide margin. Unlike I/O, the difference in CPU time between TES$_\delta$ and TES$_\lambda$ is marginal, since they share the same module for candidate set updates, which dominates the CPU overhead. Figure 8(b) also shows the number of DTop-k results with the poly-line and the vertical axis on the right. The number of results increases with k, since a larger k lowers the threshold for objects to enter the top-k set.

Figure 9 repeats the same experiments on the *API* data. The results for LHL are also included, since there are few (i.e., 14) time series, and the query windows are long; these settings favor LHL. TES is again the clear winner in all settings in terms of I/O. LHL has the lowest CPU overhead for this dataset, since scanning the compressed rank lists takes negligible time. However, its I/O cost is considerably higher than TES, and I/O is the dominating factor here, as the CPU times of all methods are below 10ms, which is equivalent to less than 2 random I/Os on our server. Hence, LHL incurs the highest overall query response time. The performance of NAI and TES leads to similar conclusions as on the *Stock* dataset, except that random fluctuations have a higher impact, due to the abnormal shape of the data, i.e., few, but very long time series. Finally, the number of DTop-k results increases with k, for the same reason as on the *Stock* data. We omit testing the effect of k on synthetic data, since the results are similar to those of *Stock*. In addition, we exclude additional experiments on *AOL* from the paper, because this dataset shows similar results as *Stock* with short query windows.

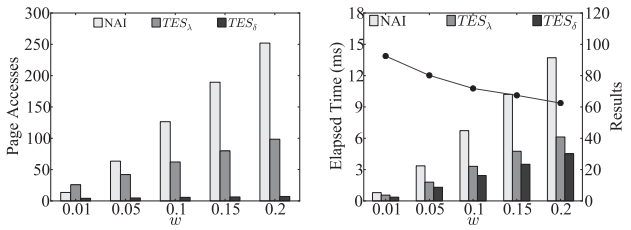


(a) IO cost (b) CPU time

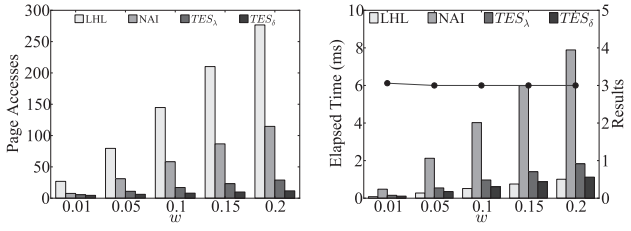
Fig. 9. Effect of k on *API*

Figure 10 investigates the effect of query window size w. The performance advantage of TES is clear in all settings. The I/O and CPU costs for all methods increase with w, since a longer query window causes more snapshot top-k retrievals in NAI and TES, and the scanning of longer portions of the rank lists in LHL. TES is generally more robust against w than

LHL and NAI in terms of I/O. The number of results decreases with w, since fewer objects exhibit durable quality over a longer time frame. Results on *Synthetic* are omitted, since they lead to similar conclusions.



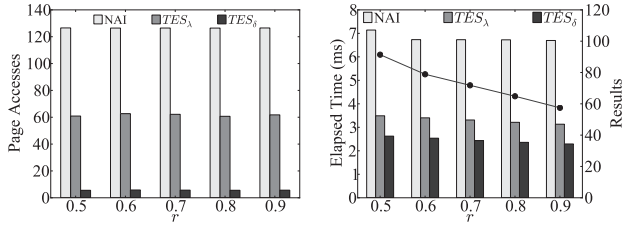
(a) IO cost (*Stock*) (b) CPU time (*Stock*)



(c) IO cost (*API*) (d) CPU time (*API*)

Fig. 10. Effect of w on real datasets

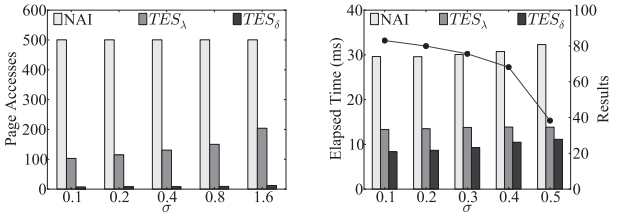
Next we focus on the impact of the durability threshold r. We show results only for *Stock* in Figure 11, which are consistent with the results on other datasets. Results for LHL are omitted since it imposes orders of magnitude higher costs. The I/O and CPU costs for all methods remain stable with different r, and their relative performance remains the same as in previous experiments. r affects the I/O cost only if it can help early termination. Larger values of r can achieve a cost decrease this way, but for the default values of the other parameters (k and w) the effect is not dramatic.



(a) IO cost (b) CPU time

Fig. 11. Effect of r on *Stock*

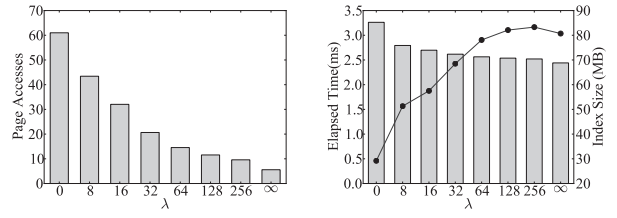
Figure 12 studies the impact of time series smoothness, using synthetic datasets generated with different values of σ, i.e., the scale of the random walk at each timestamp. Other parameters are fixed to their defaults. Increasing σ has little effect to NAI, since NAI retrieves the top-k set at each timestamp irrespective of the data volatility. On the other hand, as σ increases, more changes occur in the top-k sets between consecutive timestamps, which negatively affects TES. Still, even for the largest value of σ TES is much faster than NAI. The number of query results drops with σ as fewer sequences satisfy the durability constraint when volatility increases.



(a) IO cost (b) CPU time

Fig. 12. Effect of σ on synthetic datasets

Having established the superiority of TES over the naïve methods, we next analyze the intrinsic properties of TES. Figure 13 demonstrates the flexibility of TES_λ on the *Stock* data, with default values for k, w and r, and varying λ, i.e., the minimum length for a rank-change interval to be stored in the CJI-tree. $\lambda = 0$ corresponds to the TES_λ tested above, and $\lambda = +\infty$ reduces to TES_δ. Clearly, a small λ saves storage space, but also decreases query I/O performance. The CPU time, on the other hand, is not significantly affected by λ, indicating that the CJI-tree imposes negligible CPU overhead. In practice, the choice of λ depends on the target application; we recommend setting λ as large as the amount of storage permits to obtain high query performance.



(a) IO cost (b) CPU time

Fig. 13. Effect of λ on *Stock*

Finally, Table 6 lists the index sizes for TES_δ and TES_λ with $\lambda = 1$ on *Stock* for various values of $k_{\max}$, i.e., the maximum allowable value for k and on *Synthetic* for various values of the volatility parameter σ. With large $k_{\max}$ or σ, the index size of TES_δ can be several times higher than that of TES_λ. The difference increases with $k_{\max}$, as longer rank change intervals exist, which are replicated in TES_δ. Similarly, larger volatility increases the average length of the intervals. Nevertheless, in all settings, even for TES_δ the index size remains manageable. Considering its high query performance, TES_δ is the ideal solution for DTop-k queries in applications involving data with similar sizes as the ones used in our experiments.

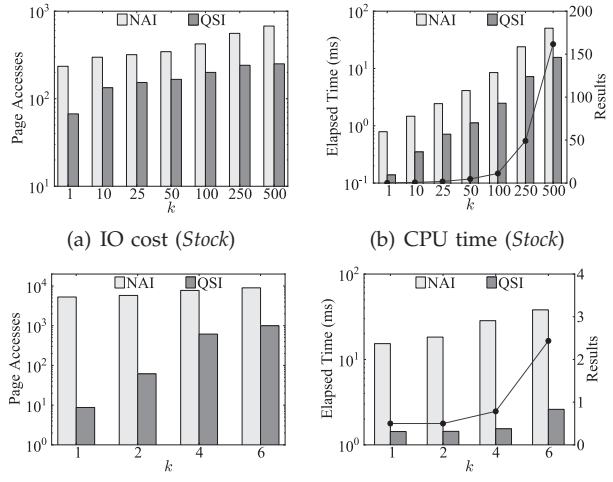
6.2 Durable k-NN Evaluation

Next we present experimental results for DkNN processing. Figure 14 shows the impact of k on the

TABLE 6
Index Size (unit: MB)

Stock k_{max}	**32**	**64**	**128**	**256**	**512**
TES_δ	0.9	3.1	9.4	27.4	80.7
TES_λ	1.0	2.9	6.9	14.6	29.2
Synthetic σ	**0.1**	**0.2**	**0.4**	**0.8**	**1.6**
TES_δ	142.5	170.8	194.2	224.3	282.3
TES_λ	110.1	111.0	110.5	110.1	110.7

I/O and CPU costs (both in logarithmic scale). The proposed algorithm QSI consistently beats the naïve method NAI, sometimes by more than an order of magnitude. The computational costs for both methods increase with k, for similar reasons as in the DTop-k experiments. With growing k, the performance gap between QSI and NAI gradually closes, because the k-NN sets at adjacent timestamps share fewer objects, forcing QSI to retrieve more data. Compared to DTop-k, there are considerably fewer DkNN results with the same parameters. The reason is that objects' ranking scores tend to be stable, whereas their similarity to a reference object exhibit significant variations at different timestamps.



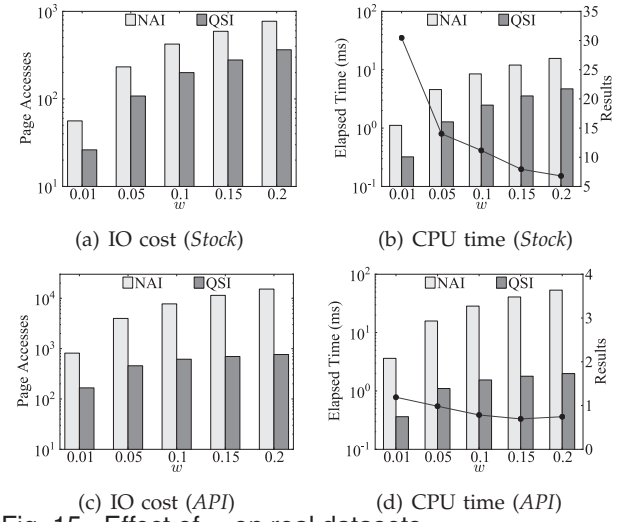
(a) IO cost (*Stock*) (b) CPU time (*Stock*)

(c) IO cost (*API*) (d) CPU time (*API*)

Fig. 14. Effect of k on real datasets

Figures 15 and 16 present the results with varying query window size w and durability threshold r, respectively. QSI is again the clear winner in all settings; the performance gap between QSI and NAI is not significantly affected by w or r. The costs of both methods increase with w, and decrease with growing r, as expected. The benefits of using QSI are more pronounced on *API* than on *Stock*, since recomputing snapshot k-NN sets in NAI is more expensive on *Stock* with a large number of series.

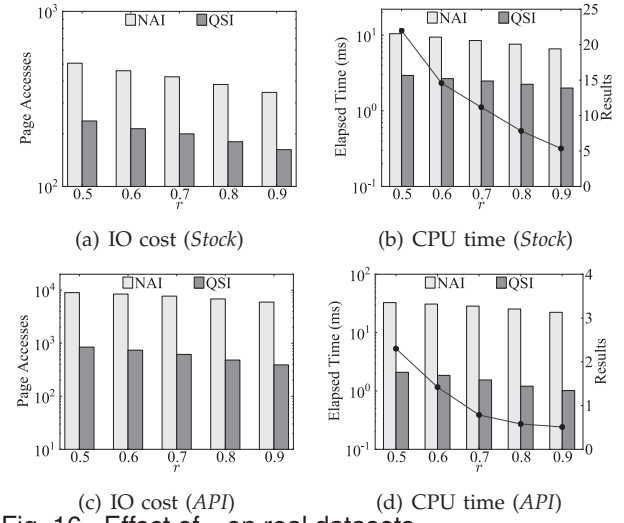
Finally, Figure 17 evaluates NAI and QSI on the synthetic data, with varying smoothness levels controlled by σ. Unlike the DTop-k results, here the I/O and CPU costs for both QSI and NAI decrease with increasing σ (meaning less smooth data). This is because as the data becomes more volatile, the number of results drops quickly. After σ reaches 0.4, the number



(a) IO cost (*Stock*) (b) CPU time (*Stock*)

(c) IO cost (*API*) (d) CPU time (*API*)

Fig. 15. Effect of w on real datasets

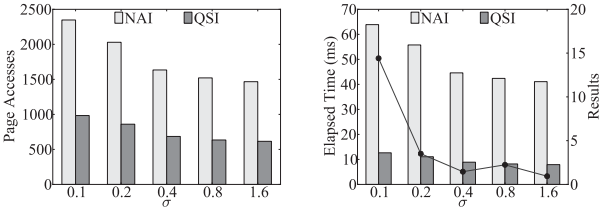


(a) IO cost (*Stock*) (b) CPU time (*Stock*)

(c) IO cost (*API*) (d) CPU time (*API*)

Fig. 16. Effect of r on real datasets

of results approaches zero, which often leads to early termination. For $\sigma > 0.4$, further increase of σ has negligible effects on query performance.



(a) IO cost (b) CPU time

Fig. 17. Effect of σ on synthetic datasets

Summarizing the experiments, the proposed methods TES and QSI significantly outperform their naïve counterparts, often by over an order of magnitude. Meanwhile, TES can be tuned to effectively balance its storage overhead and query efficiency. The costs of TES and QSI are generally low (i.e., up to hundreds of random I/Os and tens of milliseconds), suggesting that durable queries can be readily applied in practice.

7 Conclusion

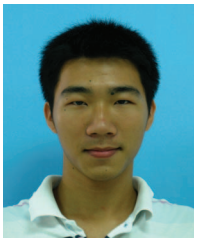
We studied the evaluation of durable queries over historical time-series databases, which find application in a variety of analytical tasks. For durable top-k queries, we proposed the TES framework, which exploits time series smoothness to reduce query costs. For durable k-NN, we developed a novel solution QSI, which indexes the query space and avoids unnecessary snapshot k-NN queries. By experimentation with both real and synthetic data, we showed that the proposed methods are very efficient compared to naïve alternatives and the previous state-of-the-art.

Currently, all proposed solutions are discussed under the assumption that each timestamp is equally important; thus, an interesting topic for future work is to extend them to handle applications with different weights on different timestamps. For example, for top-k queries, the user might be interested in objects that appear in the top-k sets frequently at the beginning or at the end of the query window. In this situation, a possible adaptation of TES would be to maintain the time-weighted sum Δ_s for each candidate object s, and devise pruning strategies accordingly. Similarly, extensions for durable k-NN queries to the time-weighted model is also an interesting direction for further investigation. Finally, we also plan to study the robustness of durable queries in the presence of noise, which is common in the time series data.

References

[1] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding-based subsequence matching of time series. In *SIGMOD*, 2008.
[2] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with Chebyshev polynomials. In *SIGMOD*, 2004.
[3] K. Chabrabarti, E. Keogh, S. Mehrotra, and M. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. *TODS*, 27(2):188–228, 2002.
[4] L. Chen, M. T. Ozsu, and V. Oria. Robust and fast similarity search for moving object trajectories. In *SIGMOD*, 2005.
[5] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *VLDB*, 2007.
[6] P. Cudré-Mauroux, E. Wu, and S. Madden. TrajStore: an adaptive storage system for very large trajectory data sets. In *ICDE*, 2010.
[7] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. 3rd Edition, Springer Verlag, 2008.
[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.
[9] R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB J.*, 19:687–714, 2010.
[10] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, 1984.
[11] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4):11:1–11:58, 2008.
[12] J. Jestes, J. M. Phillips, F. Li, and M. Tang. Ranking large temporal data. In *VLDB*, 2012.
[13] B. Jiang and J. Pei. Online interval skyline queries on time series. In *ICDE*, 2009.
[14] E. Keogh. Exact indexing of dynamic time warping. In *VLDB*, 2002.
[15] M. L. Lee, W. Hsu, L. Li, and W. H. Tok. Consistent top-k queries over time. In *DASFAA*, 2009.
[16] F. Li, K. Yi, and W. Le. Top-k queries on temporal data. *VLDB J.*, 19:715–733, 2010.
[17] C. Ré, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *ICDE*, 2007.
[18] R. Sherkat and D. Rafiei. On efficiently searching trajectories and archival data for historical similarities. *PVLDB*, 1(1):896–908, 2008.
[19] R. S. Tsay. *Analysis of Finacial Time Series*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2nd edition, 2005.
[20] L. H. U, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *SIGMOD*, 2010.
[21] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg. Reverse top-k queries. In *ICDE*, 2010.
[22] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.

Hao Wang is a PhD student at the Department of Coputer Science, University of Hong Kong (HKU). He received his BSc in Math and MSc in Computer Science from Nanjing University (NJU). Before joining HKU, he worked at Trend Micro Inc., and at the State Key Laboratory for Novel Software Technology at NJU. His research interests include ranking queries, spatio-temporal databases, and constrained optimization.



Yilun Cai is a PhD student at the Department of Computer Science, University of Hong Kong. He received his BSc and MEng in Computer Science from Sun Yat-sen University, China. His research interests include spatial and temporal query processing, text data management and mining, and query optimization in database systems.

Yin Yang is a Research Scientist at the Advanced Digital Sciences Center (ADSC), Singapore, and a Principal Research Affiliate at the University of Illinois at Urbana-Champaign. He obtained his PhD in Computer Science from the Hong Kong University of Science and Technology (HKUST). His research interests include database security and privacy, spatial and temporal query processing, and distributed systems.

Shiming Zhang received his PhD in Computer Science from the University of Hong Kong (HKU). Before coming to HKU, Zhang worked at the National Lab of Computer Science, and the Computer Science Corporation. His research focuses on spatio-temporal query processing, time series mining, finance computing, etc.

Nikos Mamoulis received a PhD in Computer Science from the Hong Kong University of Science and Technology. He is currently a professor at the University of Hong Kong. His research focuses on management and mining of complex data, privacy and security in databases, and uncertain data management. He served as PC member in more than 80 international conferences on data management and mining. He is an associate editor for IEEE TKDE and the VLDB J.