Durable Top-k Queries on Temporal Data

Junyang Gao
Duke University
jygao@cs.duke.edu

Pankaj K. Agarwal
Duke University
pankaj@cs.duke.edu

Jun Yang
Duke University
junyang@cs.duke.edu

ABSTRACT

Many datasets have a temporal dimension and contain a wealth of historical information. When using such data to make decisions, we often want to examine not only the current snapshot of the data but also its history. For example, given a result object of a snapshot query, we can ask for its "durability," or intuitively, how long (or how often) it was valid in the past. This paper considers *durable top-k queries*, which look for objects whose values were among the top k for at least some fraction of the times during a given interval—e.g., stocks that were among the top 20 most heavily traded for at least 80% of the trading days during the last quarter of 2017. We present a comprehensive suite of techniques for solving this problem, ranging from exact algorithms where k is fixed in advance, to approximate methods that work for any k and are able to exploit workload and data characteristics to improve accuracy while capping index cost. We show that our methods vastly outperform baseline and previous methods using both real and synthetic datasets.

PVLDB Reference Format:
Junyang Gao, Pankaj K. Agarwal, Jun Yang. Durable Top-k Queries on Temporal Data. *PVLDB*, 11 (13): 2223-2235, 2018.
DOI: https://doi.org/10.14778/3275366.3275371

1. INTRODUCTION

Many domains naturally produce temporal data. Making decision with such data involves considering not only the current snapshot of the data, but also its history. We consider the problem of finding "durable" objects from temporal data. Intuitively, while a snapshot query returns objects satisfying the query condition in the current snapshot, a *durability query* returns objects that satisfy the query condition with some consistency over time. Durability queries can vary in complexity. As a simple example, in an environmental monitoring setting, a scientist may want to know locations where pollutant levels have consistently remained above a threshold considered dangerous. As a more complex example, in a stock market, an investor may be interested in stocks whose price-to-earning ratios had been among the lowest 10 in the tech industry over 80% of the time over the past year.

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.

Proceedings of the VLDB Endowment, Vol. 11, No. 13
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3275366.3275371

In this paper, we tackle *τ-durable top-k queries*, which generalize the last example above and have also been considered in [19]. Given a database of objects with time-varying attributes, assume that we can rank these objects for every time instant. Intuitively, a τ-durable top-k query returns, given a query interval I, objects that rank among the top k for at least τ fraction of the time instants in I. In our last example above, $\tau = 0.8$ and $k = 10$. We give a more formal problem statement below.

Problem Definition. Consider a discrete time domain of interest $\mathbb{T} = \{1, 2, \ldots, m\}$ and a set of objects labeled $1, 2, \ldots, n$, where each object i has a time-varying value given by function $v_i : \mathbb{T} \to \mathbb{R}$. Let $\mathcal{D} = \{v_i \mid 1 \le i \le n\}$ denote this time series database.

Given time $t \in \mathbb{T}$ and object i, let $\mathsf{rank}_i(t)$ denote the rank of i among all objects according to their values at time t; i.e., $\mathsf{rank}_i(t) = 1 + \sum_{1 \le j \le n} \mathbf{1}[v_j(t) > v_i(t)]$.

Given a non-empty interval $[a, b) \subset \mathbb{T}$,[1] we define $\mathsf{dur}_i^k([a, b))$, the *durability of object i over $[a, b)$ (with respect to a top-k query)*, as the fraction of time during $[a, b)$ when object i ranks k or above; i.e., $\mathsf{dur}_i^k([a, b)) = (\sum_{t \in [a,b)} \mathbf{1}[\mathsf{rank}_i(t) \le k])/(b - a)$.

Given $\mathcal{D}$, a non-empty interval $I \subseteq \mathbb{T}$, and a *durability threshold* $\tau \in [0, 1]$, a *durable top-k query*, denoted $\mathsf{DurTop}^k(I, \tau)$, returns the set of objects whose durability during I is at least τ, i.e., $\mathsf{DurTop}^k(I, \tau) = \{i \in [1, n] \mid \mathsf{dur}_i^k(I) \ge \tau\}$.

Contributions. We present a comprehensive suite of techniques for answering durable top-k queries. First, even in the simpler case when the query parameter k is fixed and known in advance, application of standard techniques would lead to query complexity linear in either the number of objects, or the total number of times objects entering or leaving the top k during the query interval. We develop a novel method based on a geometric **reduction to *3d halfspace reporting*** [1], with query complexity only linear in the number of objects in the result, which can be substantially less than how many times they enter or leave the top k during the query interval.

When k is not known in advance, supporting efficient queries becomes more challenging. A straightforward solution is to extend the fixed-k solution and build an index for each possible k, but doing so is impractical when there are many possible k values. Instead, we consider two approaches for computing approximate answers: *sampling-based* and *index-based* approximation. The **sampling-based approach** randomly samples time instants in

[1] By abuse of notation, we use $[a, b)$ to represent all integers in interval $[a, b)$: i.e., $[a, b) = \{a, a + 1, \ldots, b - 1\}$, where $1 \le a < b \le m + 1$. We will use this notation throughout the paper without further explanation if the context is clear.

the query interval, and approximates the answer with the set of objects that are durable over the sampled time instants (instead of the full query interval). It provides a good trade-off between query time (number of samples drawn) and result quality. The **index-based approach** selects useful information to index in advance—much like a *synopsis* [4]—from which queries with any k can be answered approximately. We frame the problem of selecting what to index as an optimization problem whose objective is to minimize expected error over a query workload, and explore alternative solution strategies with different search spaces. This approach is able to achieve high-quality approximate answers with fast query time and low index space.

2. RELATED WORK

Lee et al. [11] considered the problem of computing *consistent top-k queries* over time, which are essentially a special case of τ-durable top-k queries with $\tau = 1$. The basic idea of their solution is to go through the query interval and verify membership of objects in the top k for very time instant. This process can be further sped up by precomputing the rank of each object at every time instant and storing this information in a compressed format. However, for long query intervals, this approach is still inefficient as its running time is linear in the length of the query interval (as measured by the number of time instants).

Wang et al. [19] extends the problem to the general case of $\tau \leq 1$. One key observation is that in practice, between two consecutive time instants, the set of top k objects is likely to change little. Their approach, called *TES*, precomputes and indexes changes to top-k memberships over time (and only at times when actual changes occur). Given a query interval, TES first retrieves the top k objects at the start of the interval. Next, using its index, TES finds the next time instant when the top-k set differs from the current, and updates the set of candidate objects and how long they have been in the top k so far; those with no chance of meeting the durability threshold (even assuming they are among the top k during the entire remaining interval to be processed) can be pruned. The process continues until we reach the end of the query interval. The time complexity of TES is linear in the total number of times objects entering or leaving the top k during the query interval, which can still be as high as k times the length of the query interval for complex temporal data.

Durable top-k queries also arise in informational retrieval [13]. Given a set of versioned documents (web pages whose contents change over time), a set of query keywords Q and a time interval I, the problem is to find documents that are consistently—more than τ fraction of the time over over I—among the most relevant to Q. The focus of [13] is how to merge multiple per-keyword rankings over time efficiently into a ranking for Q, based on the rank aggregation algorithm by Fagin et al. [6]. The problem in our setting does not have this dimension of Q, so we are able to devise more efficient indexes and algorithms. Finally, approximation has not been addressed by any previous work above [11, 13, 19].

Returning τ-durable top-k objects is related to ranking temporal objects based on their durability score during the query window, which leads to another line of related work on *ranking temporal data*. Li et al. [12] first considered *instant top-k queries*, which ranks temporal objects based on a snapshot score for a given time instant. Then, Jestes et al. [10] studied a more general and robust ranking operation on temporal data based on aggregation—for each temporal object, an aggregate score (based on average or sum, for example) is first computed from the object's time-varying value over the query interval, and then the objects are ranked according to these scores. Note that their problem is markedly different from ours: in our problem setting, we cannot directly compute the durability score of an object without examining all other objects' values over the query interval. Nonetheless, given a fixed k, we could precompute a time-varying quantity $h_i^k(t) = \mathbf{1}[\mathsf{rank}_i(t) \leq k]$ for each object i and treat the results as input to the problem in [10], with durability defined using the sum of $h_i^k(t)$ over time. Indeed, one of the baseline methods we consider in Section 3 for the simple case of fixed k, based on precomputed prefix sums [8], uses essentially the same idea as the exact algorithm in [10]. The case of variable k we consider requires very different approaches. While approximation was also considered in [10], they focus on approximating each object's time-varying value with selected "breakpoints" in time. In contrast, because we cannot afford to index $h_i^k(t)$ for all possible k values, we focus on how to select k's to index in Section 4.2, which is orthogonal to the approach in [10].

3. DURABLE TOP-K WITH FIXED K

This section considers the simpler case of durable top-k queries where the query parameter k is fixed and known in advance; only the query interval I is variable. Practically, this problem is less interesting than the case where k is variable and known only at query time. Nonetheless, we study this problem because its solutions can be used as a building block in solving the variable k case. We shall quickly go over two baseline methods based on standard techniques, and then present in more detail a novel method based on a geometric reduction. All these methods are exact.

Before presenting these methods, we introduce some notation. For each object i, we define the time-varying indicator function $h_i^k(t) = \mathbf{1}[\mathsf{rank}_i(t) \leq k]$ for $t \in \mathbb{T}$; its value at time t is 1 when object i is among the top k at time t, or 0 otherwise. The durability of object i over query interval I is simply the sum of this function over $t \in I$, divided by the length of I. According to this function, we can define for each object i a partitioning of $\mathbb{T}$ into a list $\mathcal{I}_i^k$ of maximal intervals, such that:

- For each $J \in \mathcal{I}_i^k$, $h_i^k(t)$ remains constant (either 1 or 0) for all $t \in J$. We call J a 1-*interval* if this constant is 1, or 0-*interval* otherwise.
- For each pair of consecutive intervals J and J' in $\mathcal{I}_i^k$, $h_i^k(t) \neq h_i^k(t')$ for all $t \in J$ and $t' \in J'$. In other words, 1-intervals and 0-intervals alternate in $\mathcal{I}_i^k$, and each of them is maximal.

Intuitively, $|\mathcal{I}_i^k|$, the number of intervals in $\mathcal{I}_i^k$, measures the "complexity" of $h_i^k(t)$ and is basically (one plus) the number of times that object i enters or leaves the top k. Give k, we write $|\mathcal{I}^k| = \sum_{i=1}^n |\mathcal{I}_i^k|$ for the overall complexity of top-k membership over time, or roughly, the total number of times that objects enter or leave the top k over time. Given time interval I, we write $|\mathcal{I}^k[I]| = \sum_{i=1}^n \sum_{J \in \mathcal{I}_i^k} \mathbf{1}[J \cap I \neq \varnothing]$ for the complexity of top-k membership over I.

Note that given k, computing $\mathcal{I}_i^k$ (equivalently, $h_i^k(t)$) for all objects takes only $O(mn)$ (i.e., linear) time, assuming that data is clustered by time such that the values of all objects at any time instant can be efficiently retrieved—even if they may not be sorted by value, a linear-time (top-k) selection algorithm can compute the membership of each object in the top k [3, 16]. If data is not clustered by time, we simply sort first. All methods below require computing $h_i^k(t)$ and/or $\mathcal{I}_i^k$ for all objects for index construction.

3.1 Baseline Methods

Prefix Sums. A simple method for finding the τ-durable top-k objects would be to compute the durability of each object over the query interval and check if it is at least τ. Instead of computing the durability an object i naively by summing $h_i^k(t)$ over the query

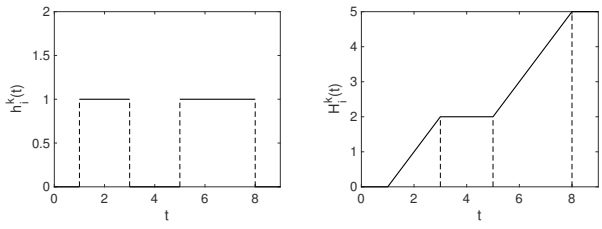


Figure 1: Example $h_i^k(t)$ (left) and $H_i^k(t)$ (right).

interval instant by instant, a standard method is to precompute and index the *prefix sums* [8] for $h_i^k(t)$, defined as follows: $H_i^k(1) = 0$ and $H_i^k(t) = \sum_{1 \le t' < t} h_i^k(t)$. Then, we can compute the durability of object i over interval $[a, b)$ using the prefix sums at the interval endpoints; i.e., $\mathsf{dur}_i^k([a, b)) = (H_i^k(b) - H_i^k(a))/(b - a)$. The prefix-sum function $H_i^k(t)$ is piecewise-linear, as illustrated in Figure 1, where each piece corresponds to a 1-interval (if the slope is 1) or 0-interval (if the slope is 0). Thus, we need to store and index only the breakpoints in a standard search tree (such as B+tree), which takes $O(|\mathcal{I}_i^k|)$ space and supports $H_i^k(t)$ lookups (and hence durability computation over any interval) in $O(\log|\mathcal{I}_i^k|)$ time, independent of the length of the query interval. The same idea was used in [10], as discussed in Section 2.

In practice, unless $|\mathcal{I}_i^k|$ is large, it is feasible to simply store $H_i^k(t)$ either as a sparse array of time-count pairs sorted by time, or as a dense array of counts (where the array index implicitly encodes the time), whichever uses less space. Doing so does not change the asymptotic space or time complexity, but often results in more compact storage.

Overall, given k, precomputing and indexing H_i^k for all objects only takes time linear in the size of the database, and requires $O(|\mathcal{I}^k|)$ index storage. With this method, although testing whether an object τ-durable is very efficient, we must still check every object, so the running time of a durable top-k query is still linear in n, the total number of objects.

Interval Index. In practice, when $k \ll n$, many objects may never enter the top k at any point during the query interval; the method above could waste significant time checking these objects. To avoid such unnecessary work, we can apply another standard technique: storing the 1-intervals for all objects in standard interval index (such as interval tree) that supports efficient reporting of intervals overlapping a query interval (logarithmic in the number of indexed intervals and linear in the number of result intervals). Given a query interval I, we use the index to find all 1-intervals that overlap with I, and simply go through these 1-intervals to compute durabilities for objects associated with these intervals (those not entirely contained in I require special, but straightforward, handling). Any object with 0 durability in I will never come up for processing.

Overall, given k, precomputing and indexing 1-intervals for all objects takes time linear in the size of the database, and requires $O(|\mathcal{I}^k|)$ index storage. The running time of a durable top-k query over interval I is logarithmic in $|\mathcal{I}^k|$ but linear in $|\mathcal{I}^k[I]|$ (or the number of times objects enter and leave top k during I).

3.2 Reduction to 3d Halfspace Reporting

The two baseline methods each have their own weakness. In practice, durable top-k queries tend to be selective—after all, they intend to find special objects. However, the method of prefix sums examines every object (and hence runs in time linear in n), while the method of interval index examines all 1-intervals during the query interval (and hence runs in time linear in $|\mathcal{I}^k[I]|$). These methods can end up examining substantially more objects beyond those in the actual result, as will shall see from experiments in Section 5.1. Ideally, we would like an algorithm whose running time is linear only in the number of actual result objects. In this section, we present a novel reduction of durable top-k queries (for a fixed k) to 3d halfspace reporting queries. Using the halfspace reporting data structure proposed in [1], we can answer a durable top-k query in time polylogarithmic in $|\mathcal{I}^k|$ plus the number of result objects.

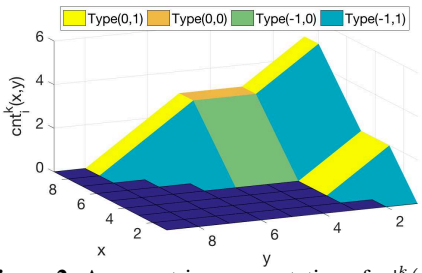


Figure 2: A geometric representation of $\mathsf{cnt}_i^k(x, y)$.

The 3d halfspace reporting problem asks to preprocess a set of points in $\mathbb{R}^3$ in a data structure such that all points lying in a query halfspace can be reported efficiently. Using the well-known point-plane duality transform [5], an equivalent formulation of the problem is to store a set of planes in $\mathbb{R}^3$ such that all planes below/above a query point can be reported efficiently.

Consider object i. Let $\mathsf{cnt}_i^k(x, y)$ be the number of times that object i ranks within the top k during $[x, y)$. We show that $\mathsf{cnt}_i^k(x, y)$ can be represented by a bivariate piecewise-linear function, with the domain of each piece being a rectangle of the form $[a, b) \times [a', b') \subseteq \mathbb{T}^2$,[2] where both $[a, b)$ and $[a', b')$ are intervals in $\mathcal{I}_i^k$ and $[a, b)$ precedes or is the same as $[a', b')$; see Figure 2. There are a total of $N = |\mathcal{I}_i^k|(|\mathcal{I}_i^k| + 1)/2$ pieces. Note that:

- If $[a, b)$ is a 1-interval, then cnt_i^k is linear in x with x-slope of -1. Intuitively, when x lies in a 1-interval, $\mathsf{cnt}_i^k(x + 1, y)$ will be one less than $\mathsf{cnt}_i^k(x, y)$, for losing the contribution of 1 from time instant x. On the other hand, if $[a, b)$ is a 0-interval, then cnt_i^k does not change with x.
- If $[a', b')$ is a 1-interval, then cnt_i^k is linear in y with y-slope of 1. Intuitively, when y lies in a 1-interval, $\mathsf{cnt}_i^k(x, y + 1)$ will be one more than $\mathsf{cnt}_i^k(x, y)$, for gaining the contribution of 1 from time instant y. On the other hand, if $[a', b')$ is a 0-interval, then cnt_i^k does not change with y.

Therefore, based on their domains, the linear functions can be classified into four types $(0, 0)$, $(0, 1)$, $(-1, 0)$, and $(-1, 1)$ below (here $c = \mathsf{cnt}_i^k(a, a')$):

	$[a', b')$ is 0-interval	$[a', b')$ is 1-interval
$[a, b)$ is 0-interval	Type $(0, 0)$: c	Type $(0, 1)$: $c + (y - a')$
$[a, b)$ is 1-interval	Type $(-1, 0)$: $c - (x - a)$	Type $(-1, 1)$: $c - (x - a) + (y - a')$

Geometrically, as Figure 2 shows, $\mathsf{cnt}_i^k(x, y)$ consists of $O(|\mathcal{I}_i^k|^2)$ rectangular pieces classified into the four types above. Now, imagine that in this 3d space, we put together all such pieces for all n objects in our database. Note that $\mathsf{DurTop}^k([x, y), \tau) = \{i \in [1, n] \mid \mathsf{cnt}_i^k(x, y) \ge \tau \cdot (y - x)\}$. From a geometric perspective, a $\mathsf{DurTop}^k([x, y), \tau)$ query is specified by a point $p = (x, y, \tau(y - x))$ in 3d, and should return precisely those pieces laying above or containing p—each such piece corresponds to a result object. With the index structure and algorithm in [2], we can support this query in $O(N \text{ polylog } N)$ space and $O(\text{polylog } N + |A|)$ time, where $N = \sum_{i=1}^n |\mathcal{I}_i^k|(|\mathcal{I}_i^k|+1)/2$, and $|A|$ denotes the number of result objects. Note that $O(\text{polylog } N) = O(\text{polylog}|\mathcal{I}^k|^2) = O(\text{polylog}|\mathcal{I}^k|)$.

[2]Here, instead of interpreting $[a, b)$ and $[a', b')$ as sets of consecutive integers as before, we treat them as continuous intervals, and $[a, b) \times [a', b')$ would technically be a rectangle in $\mathbb{R}^2$.

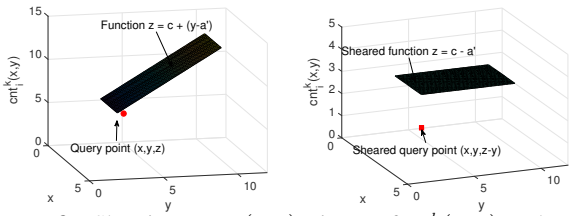


Figure 3: Shearing Type-$(0,1)$ pieces of $\mathsf{cnt}_i^k(x,y)$ to be axis-aligned. The original coordinate space is on left and the transformed space on the right.

A Practical R-tree Implementation. As practical alternative to the theoretically optimal data structure from [2], we can index all pieces of $\mathsf{cnt}_i^k(x,y)$ from all objects in a single 3d R-tree. However, an obvious shortcoming of this approach is that many such pieces—namely, those of types $(0,1)$, $(-1,0)$, and $(-1,1)$)—are not axis-aligned, so they have rather large and loose bounding boxes that lead to poor query performance.

Taking advantage of the observation that the pieces of $\mathsf{cnt}_i^k(x,y)$ have only four distinct orientations, we propose a simple yet effective alternative that avoids the problem of non-axis-aligned pieces altogether. We use four 3d R-trees, one to index each type of $\mathsf{cnt}_i^k(x,y)$ pieces for all objects. Within each R-tree, all pieces share the same orientation and have boundaries parallel to each other's, making them efficient to index as a group (more details below). Then, a $\mathsf{DurTop}^k([x,y),\tau)$ query can be decomposed into four 3d intersection queries, one for each of the R-trees.

In particular, for the R-tree indexing all Type-$(0,0)$ pieces, each piece is an axis-aligned rectangle $[a,b)\times[a',b')\times[c,c]$, lying parallel to the xy-plane and vertically positioned at c. To answer (the part of) $\mathsf{DurTop}^k([x,y),\tau)$ in this R-tree, we simply need to find all rectangles stabbed by an upward vertical ray originating from $(x,y,\tau(y-x))$.

For an R-trees indexing pieces of a type other than $(0,0)$, although the pieces are not axis-aligned to begin with, we can apply a shear transformation to the 3d coordinate space such that these pieces become axis-aligned and the query ray remains vertical. Hence, indexing and querying these sheared objects becomes exactly the same problem as in the R-tree for Type-$(0,0)$ pieces. For example, consider the following shear transformation for Type-$(0,1)$ pieces, which takes a point (x,y,z) to

$$\begin{bmatrix}1&0&0\\0&1&0\\0&-1&1\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}=(x,y,z-y).$$

Under this shear transformation, a Type-$(0,1)$ piece would become an axis-aligned rectangle $[a,b)\times[a',b')\times[c-a',c-a']$, parallel to the xy-plane and vertically positioned at $c-a'$. The query ray would originate from $(x,y,\tau(y-x)-y)$ and remain upward vertical. Figure 3 illustrates this transformation. Shear transformations for other types can be similarly defined. We summarize them below:

	Type $(0,1)$	Type $(-1,0)$	Type $(-1,1)$
Shear matrix	$\begin{bmatrix}1&0&0\\0&1&0\\0&-1&1\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\1&0&1\end{bmatrix}$	$\begin{bmatrix}1&0&0\\0&1&0\\1&-1&1\end{bmatrix}$

In sum, with four R-trees, we can process a $\mathsf{DurTop}^k([x,y),\tau)$ query as four 3d queries intersecting a vertical query ray with vertically elevated axis-parallel rectangles. The total space complexity is $O(N)$, lower than the theoretically optimal structure. While this approach no longer offers the same theoretical guarantee on the query time, it uses only a simple, standard data structure, and is very efficient in practice, as we shall see from the experimental evaluation in Section 5.1.

4. DURABLE TOP-K WITH VARIABLE K

The problem when k is variable and known only at the query time is more interesting and challenging than the case of fixed k. Naively, one could support variable k by creating an index for each possible value of k, using one of the methods from Section 3. However, doing so is infeasible when there exist many possibilities for k. As discussed in Section 2, TES, the best existing solution, indexes all top-k membership changes over time, and runs in time linear to the number of such changes during the query interval. For data with complex characteristics, TES requires a large index and still has high query complexity. In practice, users may be fine approximate answers to durable top-k queries. For example, it may be acceptable if we return a few durable top-55 objects when users ask for durable top-50 objects. Hence, in this section, we study approaches that allow us answer $\mathsf{DurTop}^k(I,\tau)$ queries with variable k approximately and efficiently, with much lower space requirement.

Our methods come in two flavors: sampling-based and index-based. The **sampling-based** approach is simple: we simply sample time instants in the query interval I randomly, and use the durabilities of objects over the sampled time instants as an approximation to their durabilities over I. The **index-based** approach aims at providing approximate answers efficiently using a small (and tunable) amount of index space—preferably small enough that we can afford to keep the entire index in memory even for large datasets. To this end, this approach intelligently chooses the most useful information to index, based on query workload and data characteristics. We note that given k and an object i, h_i^k may not be all that different from h_i^{k+1} (i.e., how object i enters or leaves top k is likely similar to how it enters or leaves top $k+1$); hence, remembering h_i^k may provide a good enough approximation to h_i^{k+1}. Furthermore, not all k's are queried with equal likelihood, and for some k's and i's, h_i^k has low complexity, and may in fact simply remains 0 throughout $\mathbb{T}$. The index-based approach uses these observations to guide its selection of what to index under a space budget. The remainder this section describes the two above approaches, with more emphasis on the index-based one.

4.1 Sampling-Based Method

Given query interval I, the sampling-based method chooses a set of time instants I_R randomly from I. With a slight abuse of notation, let $\mathsf{dur}_i^k(I_R)=(\sum_{t\in I_R}\mathbf{1}[\mathsf{rank}_i(t)\le k])/|I_R|$. For each $t\in I_R$, this method computes the top k objects at time t, and keeps a running count of how many times each object has been seen so far. After examining all I_R, the method returns the objects appearing at least $\tau|I_R|$ times, i.e., those with $\mathsf{dur}_i^k(I_R)\ge\tau$, as an approximate answer to $\mathsf{DurTop}^K(I,\tau)$. With a sufficient number of sampled time instants, we can ensure that $\mathsf{dur}_i^k(I)$ and $\mathsf{dur}_i^k(I_R)$ are close with high probability, as the following lemma shows (because of space limits, see the extended version of this paper [7] for all proofs):

LEMMA 1. *For the given parameters $\epsilon,\delta\in(0,1)$, let I_R be a set of randomly sampled time instants from I of size $\frac{1}{2(\epsilon\tau)^2}\ln(\frac{2k}{\delta\tau})$. Then for any object i, $|\mathsf{dur}_i^k(I)-\mathsf{dur}_i^k(I_R)|\le\epsilon\tau$, with probability at least $1-\delta$.*

The lemma guarantees that with sufficient samples, this method with return any object with $\mathsf{dur}_i^k(I)\ge(1+\epsilon)\tau$ with probability at least $1-\delta$; moreover, it will not return any object $\mathsf{dur}_i^k(I)<(1-\epsilon)\tau$ with probability less than δ.

The running time of this method is linear in the number of samples. However, note from Lemma 1 that this number depends only on $k,\epsilon\tau$ and δ, and not on the length of the query interval. Thus, this method shines particularly for large query intervals, compared with a naive exact method that has to examine every time instant

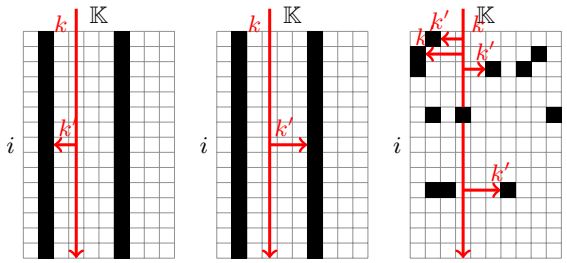


Figure 4: Illustration of three index-based methods.

in the query interval. This approach works well if data is already clustered by time, and ordered for each time instant (e.g., data on Billboard 200 music charts would be naturally organized this way). Otherwise, this approach would require either sorting objects at sampled time instants during query evaluation (which is slower) or pre-sorting objects and remembering their ordering for every time instant (which takes more space).

4.2 Index-Based Approach

We now discuss an alternative approach that indexes a small amount of data in order to answer durable top-k queries with variable k approximately. Let $\mathbb{K}$ denotes the possible values of k that can appear in a query, which in the worst case can be on the order of n, the number of objects. Indexing H_i^k (the prefix sums for h_i^k) for each object i and each $k \in \mathbb{K}$ would be infeasible. Instead, given a storage budget, we would like to choose a subset of possible (i, k) pairs and only index them instead.

In more detail, for each object i, we index H_i^k only for a subset $K_i \subseteq \mathbb{K}$. As discussed in Section 3.1, by storing the prefix sums H_i^k (which takes $|\mathcal{I}_i^k|$ space), we can compute $\mathsf{dur}_i^k(I)$ quickly using simply two fast index lookups (which takes $\log|\mathcal{I}_i^k|$ time). But what happens when the query specifies a k value not in K_i? In this case, we find some "substitute" $k' \in K_i$ where $h_i^{k'}$ "best approximates" h_i^k (we will later clarify what that means precisely later). Then, instead of checking $\mathsf{dur}_i^k(I) \geq \tau$, we would check whether $\mathsf{dur}_i^{k'}(I) \geq \tau$ to decide whether to return object i.

We introduce some additional notation before going further. Let $M \subseteq [1, n] \times \mathbb{K}$ (where $K_i = \{k \mid (i, k) \in M\}$) specify what (i, k) pairs to index. Note that we could choose $K_i = \varnothing$ for some i; in that case, we effectively "forget" object i altogether—we would pay no indexing cost for i and it would not be returned by any query. Let $\mathsf{map} : [1, n] \times \mathbb{K} \to \mathbb{K} \cup \{\perp\}$ specify the mapping function that directs queries to appropriate indexed entries. Of course, if $(i, k) \in M$, then $\mathsf{map}(i, k) = k$; otherwise, $\mathsf{map}(i, k)$ returns some $k' \in K_i$ as a "substitute." If $K_i = \varnothing$, we let $\mathsf{map}(i, k) = \perp$, $\mathcal{I}_i^\perp = \varnothing$, and $\mathsf{dur}_i^\perp(I) = 0$. The approximate answer to $\mathsf{DurTop}^k(I, \tau)$ is given by the following:

$$A^k(I, \tau) = \{i \in [1, n] \mid \mathsf{dur}_i^{k'}(I) \geq \tau \text{ where } k' = \mathsf{map}(i, k)\}.$$

We consider three different methods that follow this index-based approach. They differ in their strategy for selecting M and consequently their choice of map, as illustrated in Figure 4. Here, the candidate (i, k) pairs to be indexed are shown as square cells, and the selected ones are colored black. The simplest, **data-oblivious** method chooses the same set of k values to index across all objects, regardless of data distribution; given k, it simply maps k to the closest indexed k (for example, $k = 4$ is mapped to 2 because 4 is closer to 2 than to 7). The other two methods are data-driven in that they select their cells intelligently, based on data distribution—how much space each cell takes to index and how well it approximates nearby cells in the same row. Between these two data-driven methods, the simpler **column-wise** method limits its choices of M to columns, and its map function returns the same substitute k' for a given k consistently across all objects, like the data-oblivious method.[3] Unlike the data-oblivious method, however, its choices of M and map seek to minimize errors on the given dataset (for example, $k = 4$ may be mapped to 7 instead of 2 because it may turn out that overall H_i^7 approximates H_i^4 better than H_i^2 does across i's). The more sophisticated **cell-wise** method is free to select individual cells to index (as opposed to just columns), and its map function is "individualized" for each object (for example, given the same $k = 4$, it returns 2 for the first object, 1 for the second object, 6 for the third, etc.).

Regardless of the method for choosing M (and map), durable top-k query processing with our index-based approach is fast and has very low memory requirement. We defer the discussion of how to compute map till later when discussing each method in detail; assuming we have found the "substitute" $k' = \mathsf{map}(i, k)$, to compute $\mathsf{dur}_i^{k'}(I)$, we simply need two lookups in $H_i^{k'}$, which can be done in $O(\log|\mathcal{I}_i^{k'}|)$ time with a small, constant amount of working memory. Overall, the complexity is loosely bounded by $O(n^\star \log|\mathbb{T}|)$, where $n^\star \leq n$ is the number of indexed objects, which is no more than the number of objects that have ever entered top $\max(\mathbb{K})$. As we will see later, including the cost of computing map does not change the complexity for data-oblivious indexing and column-wise indexing, but adds $O(n^\star \log|\mathbb{K}|)$ for cell-wise indexing.

In terms of index space, as mentioned at the beginning of Section 4, our index-based approach allows the storage budget to be set as an optimization constraint. Our experiments in Section 5 show that even for large datasets (e.g., $n = 1$M and $m = 5$k, with billions of data points), to deliver fast, high-quality approximate answers, we only need a small index (e.g., a couple of GB in size) that can easily fit in main memory. If needed, our index structure also generalizes to the external-memory setting: the prefix sums and map can be implemented as B+trees; the logarithmic terms in the complexity analysis above would simply be replaced with B+tree lookup bounds.

4.2.1 *Data-Oblivious Indexing*

The data-oblivious method is straightforward. Let K denote the set of k values being indexed. We simply define $\mathsf{map}(i, k) = \arg\min_{k' \in K}|k - k'|$. By indexing the k values in K in an ordered search tree or array, we can look up $\mathsf{map}(i, k)$ for any given k in $O(\log|K|)$ time; the space is negligible. The overall index space, consumed mostly by prefix sums, is $\sum_{k \in K}\sum_{i=1}^{n}|\mathcal{I}_i^k| = \sum_{k \in K}|\mathcal{I}^k|$. We choose K such that this space does not exceed the budget allowed.

The choice of K depends on how much we know about the distribution of k in our query workload. One may choose to index the most popular k values used in queries, a geometric sequence of k values (reflecting the assumption that smaller k's are more frequently queries), or simply evenly spaced k values (reflecting the assumption that all $k \in \mathbb{K}$ are queried equally frequently), up to the budget allowed. We shall not dwell on the choice of K further here, as Section 4.2.2 below will approach this problem in a more principled manner.

4.2.2 *Data-Driven Indexing*

Before describing the two data-driven methods, we first show how to formulate the problem of choosing what to index as an optimization problem. Suppose that we know the distribution $\mathbb{Q}$ (multivariate in k, I, and τ) describing the query workload. For

[3] Although both methods index columns, for a chosen k, cells in the column corresponding to objects that never enter top k during $\mathbb{T}$ are not indexed.

simplicity, let us assume that $\mathcal{Q}$ is discrete (generalization to continuous τ is straightforward). Let $\mathbb{K} = \mathsf{supp}(\mathcal{Q}_K)$ be the support of the marginal distribution of the query rank parameter k; in other words, k will only be drawn from $\mathbb{K}$. Similarly, let $\mathbb{I} = \mathsf{supp}(\mathcal{Q}_I) \subseteq \{[a,b) \in \mathbb{T} \times \mathbb{T} \mid a \leq b\}$ be the support of the marginal distribution of the query interval parameter I. Recall that $M \subseteq [1,n] \times \mathbb{K}$ specifies the (i,k) pairs to index, and $A^k(I,\tau)$ denotes the approximate query answer computing using M and $\mathsf{map}(i,k)$. Let $\omega(M)$ denote the cost of indexing M (e.g., in terms of storage cost). Given the database $\mathcal{D}$, query workload $\mathcal{Q}$, and a cost budget B, our goal is to

$$\underset{M,\mathsf{map}}{\text{maximize}} \quad n - \mathbf{E}\Big[A^k(I,\tau) \ominus \mathsf{DurTop}^k(I,\tau)\Big] \tag{1}$$

$$\text{subject to} \quad \omega(M) \leq B. \tag{2}$$

Here, $A^k(I,\tau) \ominus \mathsf{DurTop}^k(I,\tau)$ denotes the error in the approximate answer $A^k(I,\tau)$ relative to the true answer $\mathsf{DurTop}^k(I,\tau)$, and we minimize its expectation over $\mathcal{Q}$. Note that our objective function is non-negative, since n would be the worst-case error.

The choice of the error metric $\ominus$ depends on the application. To make our discussion more concrete, here we consider the case where it computes the size of the symmetric difference between $A^k(I,\tau)$ and $\mathsf{DurTop}^k(I,\tau)$, i.e., the number of false positives and false negatives. We show how to assess this error efficiently.

Assessing Errors. Let us first break down the error (size of the symmetric difference) $A^k(I,\tau) \ominus \mathsf{DurTop}^k(I,\tau)$ by contribution from individual objects. Given k, I, τ, suppose $\mathsf{map}(i,k) = k'$. Let $\delta_i(k,k';I,\tau)$ be an indicator function denoting object i's contribution to error. Consider the two durabilities $\mathsf{dur}_i^k(I)$ and $\mathsf{dur}_i^{k'}(I)$ computed with k and k', respectively. The key observation is that object i contributes to the error only if the query threshold τ falls between these two durabilities. More precisely:

$$\delta_i(k,k';I,\tau) = \begin{cases} 1, & \text{if } \tau \in \Gamma_i(k,k';I) \\ 0, & \text{otherwise} \end{cases}$$

$$\text{where } \Gamma_i(k,k';I) = [\min\{\tau_1,\tau_2\}, \max\{\tau_1,\tau_2\}],$$
$$\text{and } \tau_1 = \mathsf{dur}_i^k(I),\ \tau_2 = \mathsf{dur}_i^{k'}(I).$$

Intuitively, $\Gamma_i(k,k';I)$ defined above establishes the "unsafe range" of τ for which error could arise: if τ is no less (or strictly greater) than both τ_1 and τ_2, then object i does not contribute to the error.

Therefore, given k and assuming $\mathsf{map}(i,k) = k'$, we can compute $d_i(k,k')$, object i's expected error contribution over $\mathcal{Q}$ (conditioned on k) as

$$d_i(k,k') = \mathbf{E}\big[\delta_i(k,k';I,\tau) \mid k\big] = \mathbf{P}\big[\tau \in \Gamma_i(k,k';I) \mid k\big]$$
$$= \sum_{I \in \mathbb{I}} \sum_{\tau \in \Gamma_i(k,k';I)} p(\tau, I \mid k).$$

Computing $d_i(k,k')$ for all possible (k,k') pairs seems daunting. However, if we assume that the distribution of k in $\mathcal{Q}$ is independent from I and τ, we can embed $\mathbb{K}$ on a line and compute d_i as simple line distance, as shown by the lemma below.

LEMMA 2. *Assume that k is independent from I and τ in $\mathcal{Q}$. Let $D_i(k) = \mathbf{P}[\tau \leq \mathsf{dur}_i^k(I)]$. Then $D_i(k)$ is non-decreasing in k, and $d_i(k,k') = |D_i(k') - D_i(k)|$.*

The lemma above implies that, we could simply precompute and store $D_i(k)$ for all $k \in \mathbb{K}$, which would allow us to compute $d_i(k,k')$ efficiently for any (k,k') pair.

Computation of $D_i(k)$'s, which is only needed at the index construction time, proceeds as follows. We first sort the entire dataset by time and value to produce the top $\max(\mathbb{K})$ objects with their ranks at each time instant. We then sort by object and time to obtain the sequence of rank changes over time for each object. After sorting, we can process each object i in turn. For each $k \in \mathbb{K}$, we scan object i's sequence of rank changes sequentially to compute the prefix sums H_i^k, which we store in memory using $O(|\mathcal{I}_i^k|)$ space. $D_i(k)$ involves summing over all possible I and τ values. With the prefix sums in memory, we can compute $\mathsf{dur}_i^k(I)$ efficiently given any I; the same $\mathsf{dur}_i^k(I)$ then allows us to evaluate predicate $\tau \leq \mathsf{dur}_i^k(I)$ for any possible τ value. Thus, the remaining expensive factor in computing $D_i(k)$ is enumerating possible I values. Fortunately, there is no need to compute $D_i(k)$'s precisely, because after all, we are simply using them to estimate error for the optimization problem. In practice, we use a Monte Carlo approach, sampling I from $\mathbb{I}$ to obtain approximate $D_i(k)$ values. Our experiments in Section 5 show that even with very low sampling rates, the approximate $D_i(k)$ values still lead to index choices that have high-quality answers.

Finally, returning to the maximization objective in (1), we have

$$n - \mathbf{E}\Big[A^k(I,\tau) \ominus \mathsf{DurTop}^k(I,\tau)\Big]$$
$$= n - \sum_{k \in \mathbb{K}} \left(p(k) \sum_{i=1}^{n} d_i(k, \mathsf{map}(i,k)) \right). \tag{3}$$

4.2.2.1 *Column-wise Indexing.*

The column-wise method makes several simplifying assumptions to make the optimization problem easier to solve. First, we restrict ourselves to selecting columns of cells from $[1,n] \times \mathbb{K}$; i.e., we pick only $K \subseteq \mathbb{K}$ for all objects and $M = [1,n] \times K$. Second, we restrict map to return the same substitute for a given k across all objects; hence, we would write $\mathsf{map}(k)$ instead of $\mathsf{map}(i,k)$. Third, we let $\omega(M) = |K|$, and we specify the budget B in terms of the number of different k values we choose to index (as opposed to a more accurate measure of index space).

Under these assumptions, we define $d(k,k') = \sum_{i=1}^{n} d_i(k,k')$ as the expected overall error in answer if we substitute k' for k. Naturally, we define $\mathsf{map}(k) = \arg\min_{k' \in K} d(k,k')$; i.e., we map k to the substitute indexed in K that minimizes the expected overall error. Now, the optimization problem becomes to

$$\underset{K}{\text{maximize}} \quad n - \sum_{k \in \mathbb{K}} \left(p(k) \min_{k' \in K} d(k,k') \right) \tag{4}$$

$$\text{subject to} \quad |K| \leq B. \tag{5}$$

Lemma 2, which applies to $d_i(k,k')$ on an individual object basis, can be readily extended to $d(k,k')$, as the following shows.

LEMMA 3. *Assume that k is independent from I and τ in $\mathcal{Q}$. Let $D(k) = \sum_{i=1}^{n} D_i(k)$. Then $D(k)$ is non-decreasing in k, and $d(k,k') = |D(k') - D(k)|$.*

Thus, we can embed $\mathbb{K}$ on a line and compute d as simple line distance. By indexing the selected k values in K in an ordered search tree or array, we can look up $\mathsf{map}(k)$ for any given k in $O(\log|K|)$ time; the space is negligible.

This observation also implies that the optimization problem in (4)–(5) for the column-wise method in has optimal substructure, as the following lemma shows.

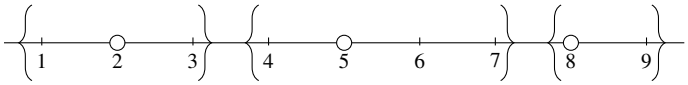


Figure 5: Partitioning of $\mathbb{K}$ by a chosen subset K. Each $k^\star \in K$ is shown as a circled point, and the interval of $\mathbb{K}$ that $k^\star$ is enclosed by { and }.

LEMMA 4. *Assume that k is independent from I and τ in $\mathcal{Q}$. Let $\mathsf{OPT}([k_1, k_2], b)$ denote the optimal solution[4] for* (4)–(5) *with $\mathbb{K} = [k_1, k_2]$ and $B = b$. Then,*

$$\mathsf{OPT}\big([k_1, k_2], b\big) = \max_{k_1 < k \le k_2} \Big\{ \mathsf{OPT}\big([k_1, k-1], b-1\big) + \mathsf{OPT}\big([k, k_2], 1\big) \Big\}$$

The above lemma immediately leads to a dynamic programming solution to the optimization problem for the column-wise method, with time complexity $O(k_{\max}^3)$, where $k_{\max} = \max(\mathbb{K})$. Note that we incur this cost only at the index construction time. The memory requirement for dynamic programming is the size of a 3d table for storing optimal substructures, which is $O(k_{\max}^3)$ in our case. In practice, $k_{\max}$ is usually not large compared with n, so we can perform dynamic programming in memory. If this 3d table is too large for memory, we can store the 3d table of optimal substructures as sequence of 2d tables organized along the dimension of budget (b). By Lemma 4, it is not hard to see that our dynamic programming procedure sequentially steps through b, so at any point during execution, we only need three 2d tables (for b, $b-1$, and 1) in memory, reducing the memory requirement to $O(k_{\max}^2)$.

Despite the simplicity of the solution, the column-wise method suffers from a rather restrictive space of possible solutions. First, `map` is not specialized for each object; even though it minimizes overall error subject to this restriction, the substitute k it produces may not be the best choice of every object. Second, indexing all entries in one single column may already take a lot of space; therefore, under tight storage constraints, the column-wise method may be forced to pick a few columns to index, hurting accuracy.

4.2.2.2 *Cell-wise Indexing.*

We now consider the more sophisticated cell-wise method, which can select any individual cells to index (as opposed to just columns) and customize its `map` function for each object. Specifically, we choose a set M of (i, k) pairs to index from $[1, n] \times \mathbb{K}$. Let $K_i = \{k \mid (i, k) \in M\}$. We define $\mathsf{map}(i, k) = \arg\min_{k' \in K_i} d_i(k, k')$, i.e., to minimize the expected error by substituting k with k' for object i. By indexing the selected k values in K_i in an ordered search tree or array, we can look up $\mathsf{map}(i, k)$ for any k in $O(\log|K_i|)$ time. Finally, we define the index storage cost as $\omega(M) = \sum_{(i,k)\in M} |\mathbb{I}_i^k|$, since storing the prefix sums for entry (i, k) takes $|\mathbb{I}_i^k|$ space (index storage for supporting `map` is negligible in comparison). The optimization problem now becomes to

$$\underset{M}{\text{maximize}} \qquad n - \sum_{i}^{n} \left(\sum_{k \in \mathbb{K}} p(k) \min_{k' \in K_i} d_i(k, k') \right) \tag{6}$$

$$\text{subject to} \qquad \omega(M) = \sum_{(i,k)\in M} |\mathbb{I}_i^k| \le B. \tag{7}$$

We show the NP-hardness of this optimization problem by reduction from the well-known knapsack problem.

LEMMA 5. *The optimization problem in* (6)–(7) *for the cell-wise method is NP-hard.*

[4]For simplicity of presentation we assume that there are no ties for the optimal solution here, but generalization to the case of ties is straightforward.

Algorithm 1: Two-phase greedy algorithm.

Input : Objective function $\mathcal{G}$ to maximize, additive cost function ω, budget B, and candidate set $U = [1, n] \times \mathbb{K}$
Output: A subset $M^\star \subseteq U$ with $\omega(M^\star) \le B$

```
S_1 ← ∅; max_1 ← 0;
foreach M ⊆ U where |M| = 1 or |M| = 2 do
    if ω(M) ≤ B then continue;
    if G(M) > max_1 then
        max_1 ← G(M);
        S_1 ← M;
S_2 ← ∅; max_2 ← 0;
foreach M ⊆ U where |M| = 3 do
    if ω(M) > B then continue;
    S ← M, I ← U \ S;
    while I ≠ ∅ do
        θ ← max_{θ∈I} (G(S ∪ {θ}) − G(S)) / ω({θ});
        if ω(S ∪ {θ}) ≤ B then S ← S ∪ {θ};
        I ← I \ {θ};
    if G(S) > max_2 then
        max_2 ← G(S);
        S_2 ← S;
if G(S_1) ≥ G(S_2) then return S_1;
else return S_2;
```

Although the problem is NP-hard, the following lemma shows that its objective function is monotone and *submodular* [15].

LEMMA 6. *The following function,*

$$\mathcal{G}(M) = n - \sum_{i}^{n} \left(\sum_{k \in \mathbb{K}} p(k) \min_{k' \in K_i} d_i(k, k') \right) \tag{8}$$

(recall $K_i = \{k \mid (i, k) \in M\}$) is a monotone and submodular set function; i.e., for all $M_1 \subseteq M_2 \subseteq [1, n] \times \mathbb{K}$ and $\theta = (i, k) \in ([1, n] \times \mathbb{K}) \setminus M_2$, we have:

$$\mathcal{G}(M_1) \le \mathcal{G}(M_2), \text{ and} \tag{9}$$

$$\mathcal{G}(M_1 \cup \{\theta\}) - \mathcal{G}(M_1) \ge \mathcal{G}(M_2 \cup \{\theta\}) - \mathcal{G}(M_2). \tag{10}$$

It was shown in [14] that a simple greedy algorithm provides a $(1 - 1/e)$-approximation for maximizing a monotone submodular set function with cardinality constraint. Sviridenko et al. further showed in [17] that a modification of the greedy algorithm for solving the problem in [14] can also produce a $(1-1/e)$-approximation for maximizing a monotone submodular set function with knapsack constraint. The modified greedy algorithm, shown as Algorithm 1, works as follows. In the first phase, we enumerates all feasible subsets of size up to two, and remember the subset S_1 that maximizes $\mathcal{G}$. In the second phase, we start with each feasible subset of size three, and try to grow it greedily and repeatedly by adding a new element at a time, which gives the largest improvement over $\mathcal{G}$ per unit cost. We remember the best subset found in the second phase as S_2. Finally, we return the better solution between S_1 and S_2.

THEOREM 1. *Let M^{opt} be the optimal solution to the cell-wise selection problem, and M^{greedy} be the solution returned by Algorithm 1. We have*

$$\mathcal{G}(M^{greedy}) \ge (1 - \frac{1}{e}) \cdot \mathcal{G}(M^{opt}).$$

In practice, enumerating all feasible subsets of size up to 3 can be expensive, so we use a simplified greedy algorithm that starts with singleton subsets and tries to grow them. It turns out that the simplified greedy algorithm still makes good choices that lead to high

Table 1: Real and synthetic datasets used in experiments.

Dataset	n (# objects)	m (# time instants)
Stock	3537	2500
Billboard	7460	1721
Temp	6756	9999
Syn/SynX	1K–10M	1K–5K

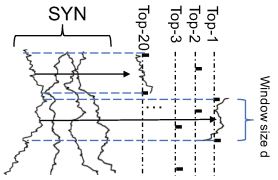


Figure 6: Generating *SynX* from *Syn*. Here, an object is boosted twice (at different times): once to top 20 and once to top 1.

query accuracy, as experiments in Section 5 shows. We also optimize the procedure for finding the next greedy choice at each step using similar techniques in [20]. We maintain a priority queue of all candidate (i, k) pairs, where $k \in \mathbb{K}$ and i is any object that has ever entered the top $\max(\mathbb{K})$. Although the memory requirement in the worst case can be $O(n|\mathbb{K}|)$, in practice only a small fraction of all objects ever enter the top $\max(\mathbb{K})$. For example, our experiments in Section 5 reveal that even for large datasets (e.g., $n = 1$M, $m = 5$k, with billions of data points and $\max(\mathbb{K}) = 10$k), the entire cell-wise optimization algorithm runs comfortably in main memory. Finally, note that we run this algorithm only at the index construction time.

5. EXPERIMENTS

In this section, we comprehensively evaluate the performance of all our methods. Section 5.1 compares the efficiency of various exact algorithms when k is fixed. For general durable top-k queries when k is variable, Section 5.2 compares various methods in terms of answer quality, query efficiency, and index space. For methods that require elaborate preprocessing and optimization for index construction, we also evaluate the efficiency of index construction.

Unless otherwise noted, all algorithms 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.

We use both real and synthetic datasets, as summarized in Table 1 and described in detail below.

Stock contains daily transaction volumes of 3537 stocks in the United States from 2000 to 2009, collected by Wharton Research Data Services.[5] We treat each stock as a temporal object, and there are 2500 time instants.

Billboard, obtained from the BILLBOARD 200 website,[6] lists weekly top-200 songs for the past 30 years. However, we are more interested in the ranking of artists as opposed to songs, as most songs remain popular only for a short duration. Thus, for our experiments, we treat artists as temporal objects, and we define the ranking of an artist in a week as the ranking of his or her top hit for that week. The result dataset has 7460 objects (artists) and 1721 time instants (weeks with rankings).

Temp, from the MesoWest project [9], contains temperature readings from weather stations across the United States over the past 20 years. For our experiments, we selected stations with sufficiently complete readings over time. The result dataset has 6756 objects (stations) and 9999 time instants (with temperature readings for all stations).

Syn refers to synthetic datasets with different sizes and distributions that we generate for in-depth comparison of various methods. Each time series is generated by an autoregressive model [18], specifically, AR(1). The model is defined by $X(t) = c + \phi X(t-1) + \epsilon(t)$, where $\epsilon(t)$ is an error term randomly chosen from a normal distribution with mean 0 and standard deviation σ, and c and ϕ are additional parameters. The mean value of the series is $\frac{c}{1-\phi}$ and the variance is $\frac{\sigma^2}{1-\phi^2}$. Tuning σ, c, and ϕ allows us to experiment with different data characteristics. For our experiments, we use $\phi = 0.6$ by default. To simulate real-life situations, we divide n objects into three groups: *elite* (20% of all objects), *mediocre* (60%), and *poor* (20%). Time series for objects from different groups are parameterized with different c values: for an elite object, we draw c from $\mathcal{N}(90, 10^2)$, normal distribution with mean 90 and standard deviation 10; for a mediocre object, $c \sim \mathcal{N}(50, 10^2)$; for poor, $c \sim \mathcal{N}(10, 10^2)$. We vary σ (in $\epsilon(t)$) from 1 to 20 for different experiments; a larger σ leads to more volatility in the time series and more rank changes.

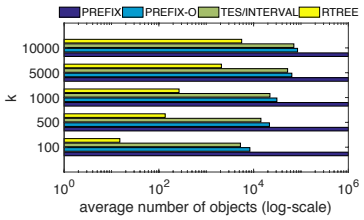
SynX is a variant of *Syn* that allows us to control the top rank changes and their durability more directly. We start with *Syn* with $\sigma = 20$ above, and establish 20 additional target values in the top range of the value domain, as shown in Figure 6. A window size parameter d controls the durability of top objects and complexity of the dataset. For each one of the 20 target values, say, v, we break the time line into $|\mathbb{T}|/d$ intervals of length d each; we vary d between 10 and 100 in our experiments. For each such interval, we randomly pick an object, and add a constant offset to its values during this interval such that the resulting average of these values becomes v—in other words, we temporarily boost the object's rank for the given interval. When picking objects to boost, we make sure that any object can be boosted at most once for any time instant.

5.1 Fixed-k Setting

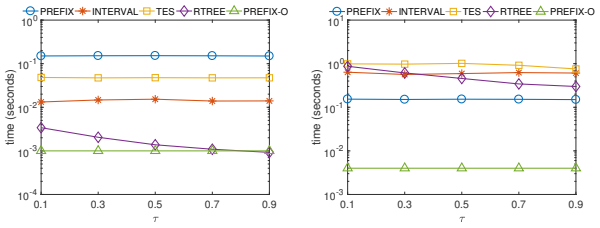
We compare five methods for answering durable top-k queries when k is fixed and known in advance: **PREFIX**, **PREFIX-O**, **INTERVAL**, **RTREE**, and **TES**. PREFIX and INTERVAL are the two baseline methods based on prefix sums and interval index, respectively, presented in Section 3.1. PREFIX-O is a variant of PREFIX with the simple optimization of not indexing an object if it is never in top k (which is also used by our index-based approach for the variable-k setting). RTREE is the practical R-tree implementation of our method based on reduction to 3d halfspace reporting, discussed in Section 3.2. TES is the state-of-the-art method from [19]. Note that TES is designed to handle variable k (up to a maximum); as k is known in this case, for a fair comparison, we optimize TES by storing rank change information only for the given k, resulting in a much simpler structure. Since all four algorithms are exact, we focus on query efficiency. We present the results of our experiments on one large synthetic dataset; additional experiments can be found in the full version [7]. Here, we use *Syn* with one million objects, five thousand time instants, and $\sigma = 10$. Results reported in Figure 7 are obtained by averaging over 1000 random durable top-k queries with randomly drawn query intervals.

Figure 7a compares the methods in terms of "pruning" power, or more precisely, how many objects they examine to answer a query (note the logarithmic scale). We set the query durability threshold $\tau = 0.2$ and try scenarios with k fixed at different values. PREFIX always needs to examine all objects. The simple optimization of PREFIX-O is surprisingly effective, and reduces the number of objects indexed and examined by one to two orders of magnitude. INTERVAL and TES have the same pruning power, as both examine objects that ever enter the top k during the query interval. Their advantage over PREFIX-O is consistent although not dramatic. Finally, RTREE reduces the number of objects considered by another one to two orders of magnitude compared with PREFIX-O/INTERVAL/TES, or up to nearly 5 orders of magnitude compared with PREFIX. Saving is bigger when k is smaller.

[5] https://wrds-web.wharton.upenn.edu/wrds/
[6] http://www.billboard.com/charts/billboard-200



(a) Objects examined per query; $\tau = 0.2$



(b) Time per query; $k = 100$

(c) Time per query; $k = 5000$

Figure 7: Comparing query efficiency for methods for the fixed-k setting. Dataset is *Syn*, with $n = 1$M, $m = 5$K, and $\sigma = 10$.

In terms of query time, however, the comparison is more nuanced. Figures 7b and 7c compare the query execution times of PREFIX, PREFIX-O, INTERVAL, TES, and RTREE, for two different settings of k, as we increase the durability threshold τ to make the queries more selective. First, in Figure 7b, where $k = 100$ is relatively small, we see that PREFIX-O and RTREE are the fastest. PREFIX-O is the fastest when queries are less selective (i.e., lower τ), but as queries become more selective, RTREE becomes faster and eventually overtakes PREFIX-O. TES is better than INTERVAL, though both pale in comparison to RTREE and PREFIX-O, and do not benefit from selective queries as RTREE does. The basic version of PREFIX is the slowest here. On the other hand, in Figure 7c, where $k = 5000$ is relatively large, we see that PREFIX-O becomes the clear winner among all methods—its performance is unaffected by the change in k. PREFIX's performance is also unchanged. However, the other methods take a hit in performance, because a larger k generally reduces opportunities for pruning (as seen in Figure 7a), so the computational overhead of pruning and more complex index structures make them less attractive, though they still examine fewer objects than PREFIX.

Overall, we conclude that PREFIX-O offers solid, competitive performance in practice, beating the theoretically more interesting RTREE except when queries are extremely selective. Because of its performance and simplicity, we also use PREFIX-O for our approximate index-based approach for handling the variable-k setting. Note that in contrast to RTREE, the pruning power of PREFIX-O heavily depends on data characteristics: for example, if every object appears in top-k at some time, no objects will be pruned, resulting in performance similar to PREFIX. However, we also note that the use of PREFIX-O by our index-based approach offers additional protection from such boundary cases, because approximation still allows us to ignore some objects that rarely ranked high, without significantly affecting accuracy.

5.2 Variable-*k* setting

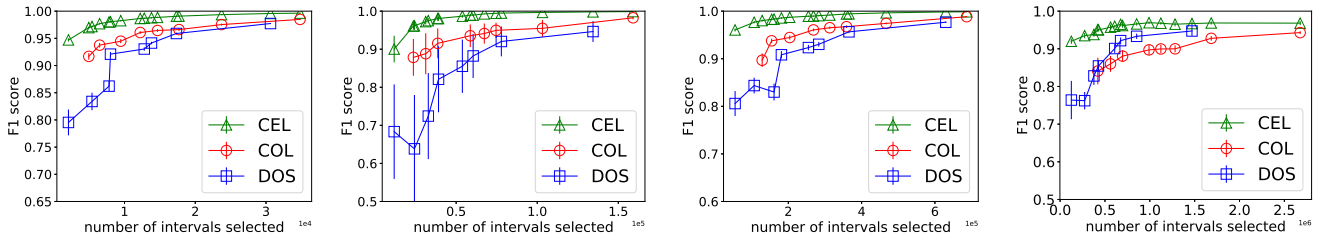
In this section, we continue to evaluate approximate methods for τ-durable top-k queries with variable k. Section 4 proposed two approaches for computing approximate answers: sample-based and index-based. Section 5.2.1 first evaluates the alternative methods for the index-based approach in terms of space and accuracy. Then, Section 5.2.2 compares the best index-based method against the sample-based approach as well as baseline and state-of-the-art approaches that produce exact answers. Finally, Section 5.2.3 evaluates the index construction costs of our index-based methods.

We use the standard F_1 score (harmonic mean of precision and recall) to measure the quality of approximate answers. The maximum possible F_1 is 1, achieved when both precision and recall are perfect. Since answer quality varies across query parameter settings, we experiment with various query workloads wherein query parameters are drawn from different distributions. Unless otherwise noted, we let $\tau = 1 - x/100$, where $\ln(x)$ is drawn from $\mathcal{N}(3, 0.5^2)$ and x is truncated to $[0, 100]$. We typically draw k from normal or log-normal distributions, discretized and truncated to appropriate ranges. Here, heavier-tailed log-normal distributions capture scenarios where users likely query with high τ and small k, but they may still try larger k or lower τ more often than a normal distribution would suggest. We typically draw the endpoints of I uniformly at random, sometimes with interval length restricted to appropriate ranges. Additional details will be given with the experiments. When constructing the indexes, our index-based methods have the knowledge of the workload distribution, but not the actual queries used in the experiments. Unless otherwise noted, for each experimental setting, we generate 1000 random queries from the workload and report both average and standard deviation for F_1 score and running time.

5.2.1 *Approximate Index-Based Methods*

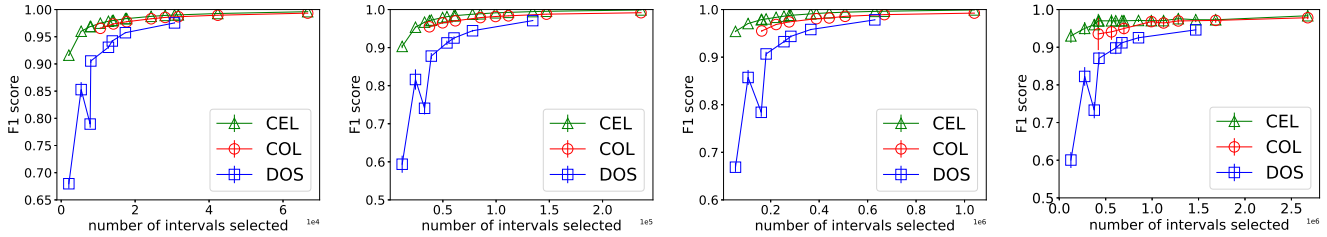
Here we compare the three index-based methods we proposed in Section 4.2: data-oblivious indexing (**DOS**), column-wise indexing (**COL**), and cell-wise indexing (**CEL**). Note that all these methods allow the index size to be adjusted, which affects their approximation quality. In the following experiments, for DOS, we generate 8 geometric sequences, with ratios 1.2, 1.4, 1.6, 1.8, 2.0, 3.0, 4.0, and 5.0. Each sequence defines a subset of columns to index in $\mathbb{K}$; e.g., ratio of 2.0 would index $k = 1, 2, 4, 8, \ldots$, up to the maximum k possible. A larger ratio implies fewer columns and hence a smaller index. For each these 8 DOS index configurations, we produce a corresponding COL index with the same number of columns (which does not guarantee the same index size, as different columns may require different amounts of index space). Finally, we use 16 actual index sizes (in terms of the number of intervals indexed)—obtained from the 8 DOS configurations and the 8 COL configurations—as constraints to produce 16 CEL configurations for comparison. Figures 8 and 9 compare the three index-based methods across four datasets, *Stock*, *Billboard*, *Temp*, and *Syn*, in terms of the quality of their approximate query answers. Results in these two figures differ in the distribution of k in the query workload—k follows a log-normal distribution in Figure 8, but a normal distribution in Figure 9; the endpoints of I are drawn uniformly at random from the time domain. As seen in both figures, CEL consistently produces answers with the highest-quality approximate answers. Even at the lowest space setting, CEL achieves F_1 scores of no less than 0.9 across datasets and query workloads. COL also offers reasonably good quality, but not as good as CEL. COL is also not as frugal as CEL or DOS in terms of space: when using the same number of columns as DOS, COL tends to consume more space.[7] DOS has unacceptably low F_1 scores at low space settings, but given more index space, F_1 scores improve, as with other two methods. In terms of the standard deviation in F_1 scores, DOS is also the worst among the three methods; CEL again is the

[7]This behavior also explains why in Figure 8d, COL does seemingly worse than DOS: given the same number of columns to index, COL in fact does offer higher accuracy than DOS, but it also chooses columns that require more space, hence pushing its curve to the right of that of DOS.



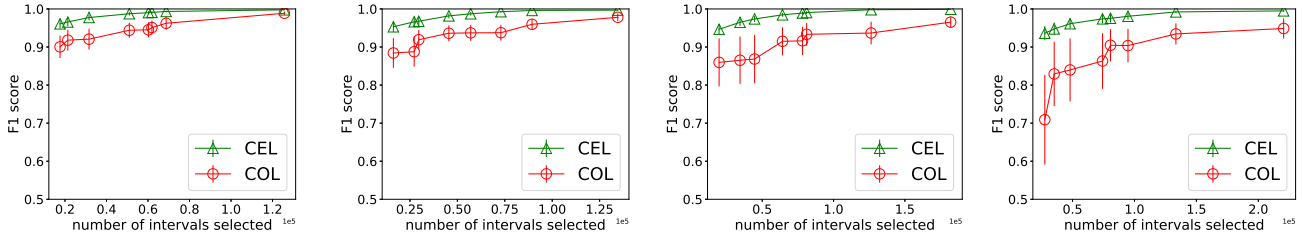
(a) *Stock* **(b)** *Billboard* **(c)** *Temp* **(d)** *Syn*, $n = 1$M, $m = 5$K, $\sigma = 10$

Figure 8: Quality of approximate answers by various index-based methods; $\ln(k) \sim \mathcal{N}(3, 0.5^2)$ and $\mathbb{K} = [1, 500]$; uniform I.



(a) *Stock* **(b)** *Billboard* **(c)** *Temp* **(d)** *Syn*, $n = 1$M, $m = 5$K, $\sigma = 10$

Figure 9: Quality of approximate answers by various index-based methods; $k \sim \mathcal{N}(50, 15^2)$ and $\mathbb{K} = [1, 500]$; uniform I.



(a) $d = 100$ **(b)** $d = 50$ **(c)** $d = 20$ **(d)** $d = 10$

Figure 10: Quality of approximate answers by CEL vs. COL on *SynX* with $n = 1$K, $m = 1$K, $\sigma = 20$; $\ln(k) \sim \mathcal{N}(3, 0.5^2)$ and $\mathbb{K} = [1, 500]$; uniform I.

best, consistently delivering high accuracy with very little variation among individual queries. Finally, between Figures 8 and 9, we see that the accuracy under DOS and COL is more sensitive to the distribution of k in the query workload than under CEL. For DOS and COL, log-normal distribution used in Figure 8 is "harder" than the normal distribution used in Figure 9, because the latter distribution is concentrated around fewer choices of k (99.7% of the density would be within $\pm 3\sigma$ of the mean), hence making it easier to pick columns to index.[8] In contrast, CEL offers consistently excellent accuracy in both figures, because it has more degrees of freedom in its choices to adapt to different query distributions.

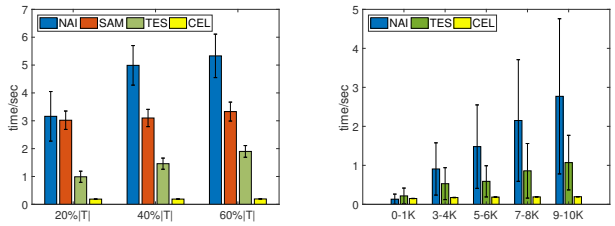
Next, we perform experiments to evaluate how well the three methods handle data with increasing complexity (in terms of rank changes over time). We use *SynX* with $\sigma = 20$ and vary d, where a smaller d leads to more frequent rank ranges. Figure 10 shows the results when k in the query workload follows a log-normal distribution (as in Figure 8); the results for normal distribution are similar and can be found in [7]. We focus on comparing just COL and CEL here because DOS is clearly inferior. In Figure 10, we see that, as d decreases and rank change complexity increases, the advantage of CEL over COL widens significantly. As complexity grows, it becomes exceedingly difficult (or simply impossible) for COL to find a set of columns and a single mapping function that work for all objects—not only doe F_1 scores drop, but the variance in F_1 scores over individual queries also increases. In contrast, CEL sees only very little degradation in F_1 score as complexity grows, and the variance remains low. For example, when $d = 10$, at the lowest space setting, CEL's F_1 score is 0.94, with a standard deviation of 0.02, compared with COL's F_1 score of 0.71 and standard deviation of 0.12. We have also experimented with instances of *Syn* with varying σ (with higher σ leading to more volatile rank changes), and drew similar conclusions; see [7] for details.

To conclude this section, CEL is the best among our index-based methods. It provides higher and more consistent accuracy across individual queries and on a wide range of datasets, and its advantages over DOS and COL become even more significant under lower space settings and for data with more complex characteristics. Another practical advantage of CEL is that it provides a smoother control over the space-accuracy trade-off than DOS and COL. DOS and COL allow the number of columns to be tuned, but some columns require more space than others to index, resulting in coarser and less predictable control over space. Moreover, DOS does not guarantee that more columns will lead to higher accuracy. In contrast, the smoother space-accuracy trade-off offered by CEL makes it easier to apply in practice.

5.2.2 *CEL vs. Other Approaches*

In this section, we compare **CEL**, our best approximate index-based method, with other approaches for answering durable top-k queries in the variable-k setting: **NAI**, **SAM**, and **TES**. NAI is a baseline exact solution, which precomputes and stores the top-$k_{\max}$

[8] An exception to this observation is that DOS has more trouble at low space settings under the normal distribution than the log-normal. The reason is that in these experiments, we hard-coded the sequences of k for DOS to index, independent of the distribution of k in the query workload; some of these sequences happen to miss the high-density region of the distribution.



(a) Varying length of I; $\mathbb{K} = [5\text{K}, 6\text{K}]$ **(b)** Varying $\mathbb{K}$; uniform I

Figure 11: Query execution times for various durable top-k solutions. *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$.

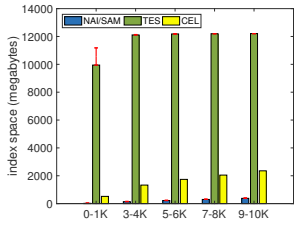


Figure 12: Index space for various durable top-k solutions. *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$; uniform I (relevant to only CEL).

membership at every time instant, where $k_{\max} = \max(\mathbb{K})$ is the maximum k that can be queried. To answer a query, NAI sequentially scans all top-k memberships in the query interval, and aggregates them to compute durability for each object it encounters. SAM is the approximate, sampling-based approach introduced in Section 4.1; it materializes exactly the same information as NAI. TES is our implementation of the state-of-the-art exact solution from [19]. Since its query performance depends on the actual data structures used, we take care to discount any possible dependency when taking measurements for TES.[9] As a result, TES query execution times reported here are only a lower bound; actual times will be higher. Moreover, although TES is intended as an external-memory solution, all indexes fit in memory in our experiments, so for a fair comparison, we implement TES using internal-memory data structures and ensure that all its data is memory-resident. For these experiments, we implemented all approaches in Python.

Note that NAI and TES are exact, while CEL and SAM are approximate. For a fair comparison, for CEL, we choose its index space budget such that CEL achieves an F_1 score of at least 0.97; for SAM, we target a similarly high accuracy guarantee with $\delta = 0.05$ and $\epsilon = 0.1$ (Lemma 1), using about 2000 samples (exact number also depends on the τ parameter in queries). We use a large synthetic dataset *Syn* with five billion data points, and compare query efficiencies for different query workloads.

Figure 11a shows how the length of the query interval I influences query execution times of various approaches. Here, we draw I's starting point randomly, and make their lengths span 20%, 40%, or 60% of the entire time domain. We draw k from $\mathcal{N}(5500, 100^2)$, truncated to $[5000, 6000]$. For NAI and TES, their execution times generally increase with the query interval length. SAM's times remain roughly the same, because the number of random samples needed is a function of the desired error bound, independent of the query interval length. Still, CEL is the fastest by a wide margin, and its times are also independent of the query interval length.

[9]TES uses a non-trivial data structure for reporting all rank changes within k during the query interval, and we do not have access to its original implementation. Hence, for the execution times of TES we report in these experiments, we simply exclude the time spent using our implementation of this data structure altogether; of course, time spent by TES processing the reported changes is still included.

Figure 11b shows how k influences the comparison of query execution times. Here, we always draw I uniformly at random, but we change the distribution of k: we start from $\mathcal{N}(500, 100^2)$ truncated to $[0, 1000]$, and then shift this distribution to the right in each setting, stopping finally at the range $[9000, 10000]$. We do not report query execution times for SAM, because for small query intervals (say, those with length less than 1000), random sampling is not applicable. From Figure 11b, we see that NAI and TES times grow roughly linearly with k; both also exhibit large standard deviations (shown as error bars), as their performance heavily depends on the query interval length. In comparison, CEL's times are consistently low (a small fraction of a second) and largely unaffected by the query parameters.

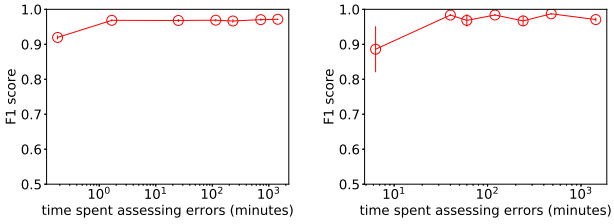
Next, we compare the index space used by the various approaches in Figure 12, as measured by the amounts of space consumed by Python data structures. The query workloads are the same as those in Figure 11b. Note that the space consumption of NAI/SAM (recall that they use the same data structure) and TES depend on the maximum k they support. Hence, for each workload setting, we report two space measurements: the higher one, shown as the red segment on top of the bar, covers the entire range of k in the workload; the lower one covers only the lower half of the range (meaning that half of the queries cannot be answered). For example, when $k \sim \mathcal{N}(500, 100^2)$ truncated to $[0, 1000]$, we report the space consumed by NAI/SAM and TES for $k_{\max} = 1000$ and $k_{\max} = 500$. CEL does not have such an issue, as it does not assume a hard limit on k. From Figure 12, we see that overall, larger k's lead to larger index space for all approaches (although CEL can operate under a specified space budget, recall that achieving the same high accuracy requires more index space for larger k's). NAI and SAM use the least amount of space, which is not surprising as these methods rely less on preprocessing. TES consumes the most space (about 13GB for $\mathbb{K} = [9000, 10000]$), which may not be acceptable as an internal-memory solution. TES's high space consumption can be explained by its approach of indexing all object rank changes over time; if data exhibit somewhat complex characteristics, indexing individual rank changes would carry a lot of overhead compared with the more compact representation of NAI/SAM. In comparison, CEL uses only 2.3GB on the highest k setting, which makes it more practical to store the index in memory. We further note that our Python-based implementation is not particularly memory-efficient. Thanks to CEL's simple data structures, a C++ implementation would reduce the memory footprint by about a factor of 2 (e.g., from 2.3GB to 1.18GB), where we can store each time instant or prefix sum with exactly 4 bytes, incurring far lower overhead than Python's implementation of lists of integers.

To further demonstrate scalability of CEL, we test it on an even larger version of *Syn* with 50 billion data points ($n = 10\text{M}$, $m = 5\text{K}$, and $\sigma = 10$). In the query workload, I is uniform and $\mathbb{K} = [5000, 6000]$. Under this setting, CEL only needs 3.6GB of index space to deliver F_1 scores of at least 0.97, with mean query execution time of 0.53 seconds (and a 0.03 standard deviation).

To summarize, CEL is both much faster and more space-efficient than TES. Even for large datasets with billions of data points, CEL only needs a couple of GB of memory to deliver fast, highly accurate results. While NAI and SAM require less space, their query execution times are not competitive.

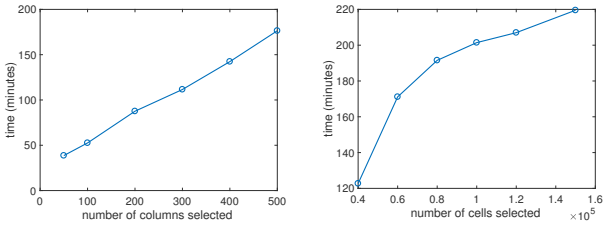
5.2.3 *Index Construction*

The two data-driven index-based methods, COL and CEL, perform elaborate preprocessing and optimization during their index construction step. In this section, we evaluate the performance of index construction for these two methods and demonstrate their



(a) *Temp* **(b)** *Syn*, $n = 1$M, $m = 5$K, $\sigma = 10$

Figure 13: CEL index quality as a function of optimization time spent on assessing expected error using Monte Carlo simulations during optimization; same query workload as Figure 8.



(a) Dynamic programming for COL **(b)** Greedy for CEL

Figure 14: Optimization time as a function of budget. *Syn*, $n = 1$M, $m = 5$K, $\sigma = 10$, $\mathbb{K} = [9\text{K}, 10\text{K}]$

feasibility on large temporal datasets. Recall that in order for these methods to select what to index, they need to 1) estimate expected error over the query workload, and 2) search for the optimal index that minimize this error under a space constraint. Both tasks can be expensive. We now take a closer look at these tasks before examining the end-to-end index construction cost.

As discussed in Section 4.2, one of the ideas we use to speed up the task of error estimation is Monte Carlo simulation, which samples from the query interval distribution to estimate the expected error. To evaluate the effectiveness of this strategy, we vary the number of samples drawn by the Monte Carlo simulation, which translates into varying index construction times; intuitively, more samples and longer running times produce more accurate estimates, which can potentially lead to higher index quality. Figure 13 shows how index quality is affected by the time spent on assessing errors (controlled by the number of Monte Carlo samples) during optimization. We show results for CEL on *Temp* and *Syn* (with five billion data points); results on other datasets and for COL are similar. The query workload is the same as in Figure 8. We measure the index quality by the observed F_1 scores on 1000 random queries generated from the workload. For a fair comparison across settings, we always give the optimization procedure the same space budget (used by the longest time settings in Figure 13 to produce a sufficiently high F_1 score). Under settings with shorter times, less accurate error estimates can potentially make the optimization procedure pick suboptimal indexes under the same space budget. As shown in Figure 13, however, even when at fairly low sampling rates—which translate to under 1.67 minutes spent on assessing errors for *Temp* or under 40 minutes for the much bigger *Syn*—we are able to deliver CEL indexes with qualities comparable to those obtained under the longest time settings. In other words, the Monte Carlo approach is quite effective in taming the cost of assessing errors while ensuring the resulting index quality.

Next, we examine the costs of the optimization algorithms: dynamic programming for COL (Section 4.2.2.1) and greedy for CEL (Section 4.2.2.2). Figure 14 plots the optimization times of COL and CEL (excluding time spent on computing error metrics) as functions of budget. The budget is in terms of the number of indexed columns for COL, and in terms of the number of indexed cells for CEL. The underlying dataset is *Syn* ($n = 1$M, $m = 5$K, $\sigma = 10$). We further stress-test index construction by enlarging the range of parameter k, drawing it from $\mathcal{N}(9500, 100^2)$, discretized and truncated to $[9000, 10000]$. Compared with the experiments on real datasets, the increases in the size of *Syn* and effective k value range together give a multiplicative boost in the search space for CEL's greedy selection algorithm, resulting in a much more challenging optimization problem. The other query workload settings are again the same as those for Figure 8. As we can see in Figure 14, generally speaking, bigger space budgets result in longer optimization times, and CEL optimization is more expensive than COL optimization. At a moderate budget settings for CEL, shown as the third data point in Figure 14(b), the resulting indexes already have F_1 scores of no less than 0.97, and require about 3.5 hours of optimization time, which is practically feasible since it only happens during index construction.

Table 2: End-to-end CEL index construction times on various datasets.

Stock	*Billboard*	*Temp*	*Syn* $m \times n = 5$ billion	*Syn* $m \times n = 50$ billion
6.05 minutes	38.56 minutes	1.97 hours	8.33 hours	16.3 hours

Finally, Table 2 lists the end-to-end CEL index construction times for all our real datasets and two large synthetic datasets. For *Stock*, *Billboard* and *Temp*, we use the same query workload as in Figure 8. For *Syn* with 5 billion data points ($n = 1$M, $m = 5$K, $\sigma = 10$), we use the query workload as in Figure 14. For *Syn* with 50 billion data points ($n = 10$M, $m = 5$K, $\sigma = 10$), we use the same query workload as the one used for this dataset in Section 5.2.2. As shown in Figure 2, for real datasets, index construction can be completed within a couple of hours. For the first *Syn* dataset, we can construct the index within 9 hours. For the second *Syn* dataset that is 10 times bigger, we can construct the index under 17 hours. Even for such large datasets, index construction time is acceptable considering that it is a one-time cost. For all datasets, the constructed CEL index provides an F_1 score of no less than 0.97.

6. CONCLUSION

In this paper, we have studied the problem of finding durable top-k objects in large temporal datasets. We first considered the case when k is fixed and known in advance, and proposed a novel solution based on a geometric reduction to the 3d halfspace reporting problem. We then studied in depth the general case where k is variable and known only at query time. We proposed a suite of approximate methods for this case, including both sampling- and index-based approaches, and considered the optimization problem of selecting what to index. As demonstrated by experiments with real and synthetic data, our best approximate method, cell-wise indexing, achieves high-quality approximate answers with fast query time and low index space on large temporal datasets.

7. ACKNOWLEDGMENTS

This work was supported by NSF grants IIS-14-08846, IIS-17-18398, IIS-18-14493, CCF-13-31133, and CCF-15-13816, an ARO grant W911NF-15-1-0408, a grant from the Knight Foundation, and a Google Faculty Award. 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.

8. 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*, pages 180–186. Society for Industrial and Applied Mathematics, 2009.

[2] P. K. Agarwal, S.-W. Cheng, and K. Yi. Range searching on uncertain data. *ACM Transactions on Algorithms (TALG)*, 8(4):43, 2012.

[3] M. Blum, R. W. Floyd, V. R. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[4] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[5] M. de Berg, O. Cheong, M. J. van Kreveld, and M. H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

[6] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of computer and system sciences*, 66(4):614–656, 2003.

[7] J. Gao, P. K. Agarwal, and J. Yang. Durable top-k queries on temporal data. Technical report, Duke University, 2018. `http://db.cs.duke.edu/papers/2018-GaoAgarwalYang-durable_topk.pdf`.

[8] C.-T. Ho, R. Agrawal, N. Megiddo, and R. Srikant. *Range queries in OLAP data cubes*, volume 26. ACM, 1997.

[9] J. Horel, M. Splitt, L. Dunn, J. Pechmann, B. White, C. Ciliberti, S. Lazarus, J. Slemmer, D. Zaff, and J. Burks. Mesowest: Cooperative mesonets in the western united states. *Bulletin of the American Meteorological Society*, 83(2):211–225, 2002.

[10] J. Jestes, J. M. Phillips, F. Li, and M. Tang. Ranking large temporal data. *PVLDB*, 5(11):1412–1423, 2012.

[11] M. L. Lee, W. Hsu, L. Li, and W. H. Tok. Consistent top-k queries over time. In *International Conference on Database Systems for Advanced Applications*, pages 51–65. Springer, 2009.

[12] F. Li, K. Yi, and W. Le. Top-k queries on temporal data. *The VLDB Journal*, 19(5):715–733, Oct. 2010.

[13] N. Mamoulis, K. Berberich, S. Bedathur, et al. Durable top-k search in document archives. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 555–566. ACM, 2010.

[14] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294, 1978.

[15] A. Schrijver. *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media, 2003.

[16] J. F. Sibeyn. External selection. *J. Algorithms*, 58(2):104–117, 2006.

[17] M. Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Operations Research Letters*, 32(1):41–43, 2004.

[18] R. S. Tsay. *Analysis of financial time series*, volume 543. John Wiley & Sons, 2005.

[19] H. Wang, Y. Cai, Y. Yang, S. Zhang, and N. Mamoulis. Durable queries over historical time series. *IEEE Transactions on Knowledge and Data Engineering*, 26(3):595–607, 2014.

[20] Y. Wu, J. Gao, P. K. Agarwal, and J. Yang. Finding diverse, high-value representatives on a surface of answers. *PVLDB*, 10(7):793–804, 2017.