Durability Queries on Temporal Data

by

Junyang Gao

Department of Computer Science
Duke University

Date: ____________________________

Approved:

____
Jun Yang, Advisor

____
Pankaj K. Agarwal

____
Ashwin Machanavajjhala

____
Sudeepa Roy

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2020

Abstract

Durability Queries on Temporal Data

by

Junyang Gao

Department of Computer Science
Duke University

Date: ____________________

Approved:

____
Jun Yang, Advisor

____
Pankaj K. Agarwal

____
Ashwin Machanavajjhala

____
Sudeepa Roy

An abstract of a dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2020

Abstract

Temporal data is ubiquitous in our everyday life, but tends to be noisy and often exhibits transient patterns. To make better decisions with data, we must avoid jumping to conclusions based on certain particular query results or observations. Instead, a useful perspective is to consider "durability", or, intuitively speaking, finding results that are robust and stand "the test of time". This dissertation studies durability queries on temporal data that return durable results efficiently and effectively.

The focus of this dissertation is two-fold: (1) design meaningful and practical notions of durability (and corresponding queries) on different types of temporal data, and (2) develop efficient techniques for durability query processing. We first study sequence-based temporal datasets where each temporal object has a series of values indexed by time. Durability queries ask for objects whose (snapshot) values were among the top k for at least some fraction of the times during a given time interval; e.g., "from 2013 to 2016, United Airlines has the highest stock price among American-based airline companies for at least 80% of the time." Second, we consider instant-stamped temporal datasets where each data record is stamped by a time instant. Here, durability queries look for records that stand out among nearby records (defined by a time window) and retain their supremacy for a long period of time; e.g. "On January 22, 2006, Kobe Bryant dropped 81 points against Toronto Raptors, a scoring record that since then has yet to be broken." Finally, going beyond analyzing historical data, we investigate the notation of durability into the future, where durability needs to be predicted by performing stochastic simulation of temporal models. For answering durability queries across these problem settings, we apply principled approaches to design fast, scalable algorithms and indexing methods. Our solutions broadly combine geometric, statistical, and approximate query processing techniques to provide a meaningful balance between query efficiency and result quality, along with theoretical worst-case (or average-case) guarantees.

To my wife, Yunqi.

To my parents, Xue and Wendong.

For their endless love, support and encouragement.

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my deepest appreciation to my advisor, Professor Jun Yang, for his guidance, support, and encouragement through each stage of my PhD studies. It has been an amazing five years working with him. I would also like to extend my gratitude to Professor Pankaj K. Agarwal for being a great mentor. Jun and Pankaj's advice has always been insightful and prompt. I'm sincerely grateful for their patience in shaping my taste of scientific research from the ground up, and in showing me the right direction towards conducting rigorous research work. They have always been the best role models for me, and will be constantly inspiring me and motivating me as an independent researcher in the future.

Many thanks to Professor Ashwin Machanavajjhala and Professor Sudeepa Roy for their service on my committee, and for their invaluable suggestions on my research work and presentations.

Thanks should also go to Xin Luna Dong, Xian Li and Yifan Xu for hosting me at Amazon Inc. for a fruitful internship during the beautiful Seattle summer in 2018. I have had great pleasure of working with them, and learning how to conduct impactful research in an engineering-research environment. I also want to thank Xiaomeng Ban and You (Will) Wu from Google Inc. for their support and guidance throughout my internship at Google in the summer of 2019. I wholeheartedly appreciate to all of you, broadening and deepening my understandings of the industrial world and academia world.

Many people played a decisive role in journey of my PhD studies. Special thanks to Professor Guoliang Li for his guidance when I was an undergraduate student in Tsinghua University. I thank him for inspiring me and leading me into the database community. Thanks to the staff at Duke, especially Marilyn Butler, for their help with all kinds of logistics, and keeping my life at Duke always well-prepared and organized. I want to thank my fellow graduate students and friends at Duke. I thank my office mate, Yuan Deng, for being a wonderful comrade along the way for the past five years. Thanks to all my friends, to name a few, Yan Chen, Zhengjie Miao, Stavros Sintos, Yuchao Tao, Yuhao Wen, Shengbao Zheng and Yuanjun Yao, for your kindness and support.

I acknowledge the funding that I received from all parties. Thank the Duke Graduate School, and the National Science Foundation.

Finally, I would like to dedicate this dissertation to my wife, Yunqi Li, and to my parents, Xue Yang and Wendong Gao. I could not appreciate more for their endless support and encouragement. None of this would have been possible without their unconditional love in me.

1

Introduction

Temporal data is ubiquitous in our everyday life, and looking forward, we can expect more and more historical data to become available, as our economic, social and scientific activities are increasingly captured in digitized forms. Given the abundance of historical data and the underlying temporal information, a useful way of working with temporal data is to consider "durability", or intuitively speaking, finding results that are robust and stand "the test of time". Real data tends to be noisy and often exhibits transient patterns. Incorporating durability on temporal data would help to rule out noises and coincidences, and lead to better decision-making with robustness and confidence. In addition, durability naturally can be interpreted into many good qualities such as consistency, uniqueness and interestingness, etc. The notions of durability are broadly applied in practice, especially in the field of media and marketing (to make eye-catching headlines). Here are a few examples where our understanding of statements based on temporal data can be strengthened by integrating durability.

Example 1. "United Airlines has the highest stock price among American-based airline companies on 2015." To show that 2015 is not just an outlier for United

Airlines, we can add that: “From 2013 to 2016, United Airlines has the highest stock price among American-based airline companies for at least 80% of the time.” Durability reflects consistency over time. The latter statement apparently conveys stronger messages for investors.

Example 2. “Kobe Bryant dropped 81 points in the game against Toronto Raptors on Jan. 22, 2006.” While impressive by itself, this statement can be boosted by adding a temporal context: “At that time, this record was the top-1 scoring performance in the past 45 years of NBA history.” Naturally, the further back we can extend the “durability” (while the record still remains top), the more convincing the statement becomes. We can extend durability forward in time as well: “Since 2006, Kobe’s 81 points scoring performance has yet to be broken as of today.” This kind of claims are popular in sports domain, and people are fascinated with those long-lasting or even “unbreakable” records.

Example 3. “On Jan. 17, 2020, Google became the third publicly traded US tech company to reach one trillion market capitalization.” Google is winning at the moment, but how about the future? One can predict durability into the future: “In the next ten years, what is the probability that Google will remain as one of the top 3 most valuable US tech companies?” Durability here shows how long (or, how likely) a condition (that currently holds) remains valid looking forward into the uncertain future.

Examples above show how durability helps make more meaningful decisions from data, discover interesting facts in data, or make robust predictions against uncertain future. These examples also illustrate many variants of the notions of durability. As in Example 1, stock market is a sequence-based temporal dataset where each temporal object (i.e., company) has a series of values indexed by time (i.e., daily stock price). Under this setting, we consider durability as a fraction of the times (during a given time interval) that an object whose (snapshot) values were among the top k. In Example 2, NBA players' scoring records is an instant-stamped temporal dataset where each record is an individual entity [1] and is stamped by a time instant. Here, durability is defined as the length of a time window where a record stands out and retains its supremacy among other records in this time window. As Example 3 shows, durability not only can be measured on existing historical data, which is certain, but also can be applied to probabilistic data based on predictions or stochastic models. It reflects the probability of a certain condition remaining valid in the future. We acknowledge that there exists multiple interpretations of durability according to different data models and application scenarios. This dissertation mainly focuses on these three types of durability as mentioned above. We hope our work could encourage researchers to explore more possibilities and interesting problems along the line of durability on temporal data.

Given the notion of durability, users can ask what we will call "durability queries" to find results with long durability from temporal data. Many durability queries are fundamentally hard and tend to be complex in nature. As the above examples show, it requires different combinations of selection, ranking and aggregation. It is challenging to develop efficient algorithms that answer durability queries, especially the ones with good theoretical performance guarantees in the worst case.

Contributions. This dissertation serves as a first step to systematically study durability queries on temporal data. We identify three types of durability queries that are popular in real-life applications, and develop a comprehensive suite of computational techniques for efficiently answering these queries. Our algorithms run significantly faster on most real temporal datasets compared to existing solutions, and are also equipped with sound worst-case performance guarantees. It allows the

[1] Though some records belong to the same player, we still consider each one of them as individual game performance.

general public to reason about temporal data incorporating durability at increased efficiency and effectiveness.

The rest of this dissertation is organized as follows.

In Chapter 2, we briefly review the previous studies regarding the notion of durability and durability queries on temporal data. More detailed literature reviews can be found in Related Work sections of Chapter 3, Chapter 4 and Chapter 5.

In Chapter 3, we first study durability queries on sequence-based temporal data, which search for objects whose values were among the top k (based on snapshot values at each timestamp) for at least some fraction of the times during a given interval. A concrete instance is shown in Example 1. We present a suite of algorithmic techniques for solving this problem, ranging from exact solutions 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. This chapter is based on joint work with Pankaj K. Agarwal and Jun Yang ([41] and [42])[2], and is reprinted with permission.

In Chapter 4, we consider durability queries on instant-stamped temporal data. Under this setting, a useful way of finding interesting or exceptional records is to explore how well they compare with other records that arrived nearby (defined by a time window), and how long they retain their supremacy. In general, given a sequence of instant-stamped records, suppose that we can rank them by a user-specified scoring function f, which may consider multiple attributes of a record to compute a single score for ranking. More formally, durability queries find records whose scores were within top k among those records within a "durability window" of given length, e.g., a 10-year window starting/ending at the timestamp of the record. The parameter k, the length of the durability window, and the parameters of the

[2] Junyang Gao is the lead author of the work, who designed, directed and coordinated the research, and provided conceptual and technical contributions for all aspects of the work.

scoring function (which capture user preference) can all be given at the query time. We propose new algorithms for solving this problem, and provide a comprehensive theoretical analysis on the complexities of the problem itself and of our algorithms. This chapter is based on joint work with Stavros Sintos, Pankaj K. Agarwal and Jun Yang ([44])[2], and is reprinted with permission.

Finally, we study durability queries on probabilistic temporal data and manage to predict durability in the future with reliability in Chapter 5. Intuitively, the notion of durability predicts how long (or, how likely) a condition that currently holds will remain valid in the future. To handle temporal data uncertainty (especially considering temporal dependence), we assume there exists a stochastic process (i.e., statistical temporal model or neural networks) that provides step-by-step predictions of a temporal series into the future. Our solution adopts a Monte Carlo approach to manage probabilistic temporal data and answer durability queries by simulating multiple possible worlds. Yet going beyond the standard Monte Carlo approach, we propose to apply a novel sampler and estimator, what we will refer to as "Multi-Level Splitting Sampling", that combine the idea from branching process theory [53] and importance sampling [27, 47, 17]. Our proposed solution is proved to provide significant simulation cost reduction compared to standard techniques in a variety of tasks in practice – up to an order-of-magnitude query time speedup without loss of answer quality. This chapter is based on joint work with Pankaj K. Agarwal and Jun Yang.[2]

We conclude the dissertation in Chapter 6, and discuss some open questions and future work.

2

Related Work

There is a connection between durability queries and queries in temporal databases [103] where tuples in database or results generally carry "validity" intervals. The database community has been devoted to studying temporal databases and its corresponding queries in the past, ranging from temporal data warehousing [113], temporal information retrieval [94], to temporal aggregations [117, 114]. As a comparison, in this dissertation, we formulate the notion of durability as a general concept on a variety types of temporal data, and durability queries we considered also tend to be more diverse and more complex in nature, as examples in Chapter 1, requiring different combinations of selection, ranking, aggregation, and even predictions over probabilistic data.

Several works have independently studied similar queries on temporal data[1], which can all be viewed as specific instances or variants of durability queries. For example, Lee et al. [73] firstly considered the problem of computing consistent top-k queries over time, which are essentially a special case of durability queries as in Ex-

[1] Detailed literature review can be found in each Chapter. Here we only sketch the high-level description of related work to give readers a broader perspective of the topic.

ample 1 and will be soon introduced in Chapter 3. Then, Wang et al. [108] further extend the problem to the more general case as our setting. In another line of work, the notion of durability is incorporated implicitly in different forms of presence. For example, in [65] and [118], authors consider the notion of durability in the form of prominent streaks in sequence data. A prominent streak is a long consecutive subsequence consisting of only large (small) values. Similarly, Jiang and Pei [64] studied Interval Skyline Queries on time series, where durability (segments of time series dominate others) is used as a dimension for skyline comparison. In this dissertation, our goal is not simply to solve different types of durability queries. In general, the key contribution of the proposed work, which sets it apart from previous studies, is the demonstration of how to leverage the special semantics of time to offer richer queries, more meaningful results, and better algorithms.

Going beyond traditional temporal data, durability queries also arise in a variety of different domains. In the field of Information Retrieval, authors [78] have studied the problem of finding versioned documents (i.e., web pages whose contents change over time) that are consistently related to given keyword(s) search during a time period. For dynamic graphs or temporal graphs, where graphs evolve over time and are typically represented as a sequence of graph snapshots, [96] and [97] have studied the problem of finding the (top-k) most durable (i.e., the longest existence) matches of an input graph pattern query. More broadly, persistent homology in computational topology [37] similarly represents the notion of durability as in temporal data. The intuition is that more persistent features (detected over a wide range of spatial scales) are more likely to represent true features of the underlying space.

3

Durable Top-k Queries on Sequence-based Temporal Data

Time is the ultimate critic.

Stewart Stafford

3.1 Introduction

Example 1 demonstrated the usefulness and interestingness of a specific type of durability queries on sequence-based temporal data. Generally speaking, let us consider a database of objects with time-varying attributes; i.e., a sequence of values over time. At each time instant, we can find a subset of objects satisfying certain query conditions in the current snapshot. To reflect durability, we could aggregate those snapshot query results and return objects that satisfy the query condition with some consistency over time. More specifically, given a period of time, a durability query searches for objects that satisfy the query condition for at least a specific number of time instants in the given time window.

In this chapter, we tackle "τ-durable top-k queries", which instantiate the (snapshot) query condition as ranking/top-k, and have also been previously considered in [108]. 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 $\mathcal{D} = \{v_i \mid 1 \leqslant i \leqslant n\}$ 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., $\mathsf{rank}_i(t) = 1 + \sum_{1 \leqslant j \leqslant n} \mathbf{1}[v_j(t) > v_i(t)]$.

Given a non-empty interval $[a, b) \subseteq \mathbb{T}$, 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., $\mathsf{dur}_i^k([a, b)) = (\sum_{t \in [a,b)} \mathbf{1}[\mathsf{rank}_i(t) \leqslant k])/(b-a)$.

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., $\mathsf{DurTop}k(I, \tau) = \{i \in [1, n] \mid \mathsf{dur}_i^k(I) \geqslant \tau\}$.

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 [3], 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 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 [29]—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.

3.2 Related work

Lee et al. [73] 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. [108] extends the problem to the general case of $\tau \leqslant 1$. 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 [78]. 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 [78] 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. [39]. 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 [73, 78, 108].

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. [76] first considered instant top-k queries, which ranks temporal objects based on a snapshot score for a given time instant. Then, Jestes et al. [63] 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 $h_i^k(t) = \mathbf{1}[\mathsf{rank}_i(t) \leqslant k]$ for each object i and treat the results as input to the problem in [63], with durability defined using the sum of $h_i^k(t)$ over time. Indeed, one of the baseline methods we consider in Section 3.3.1 for the simple case of fixed k, based on precomputed prefix sums [54], uses essentially the same idea as the exact algorithm in [63]. The case of variable k we consider requires very different approaches. While approximation was also considered in [63], 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 3.4.2, which is orthogonal to the approach in [63].

3.3 Durable Top-k Queries 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 $h_i^k(t) = \mathbf{1}[\mathsf{rank}_i(t) \leqslant k]$ 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 [16, 100]. 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.

3.3.1 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 interval instant by instant, a standard method is to precompute and index the prefix sums [54] for $h_i^k(t)$, defined as follows: $H_i^k(1) = 0$ and $H_i^k(t) = \sum_{1 \leqslant t' < t} h_i^k(t)$. 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 3.1, 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 [63].

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

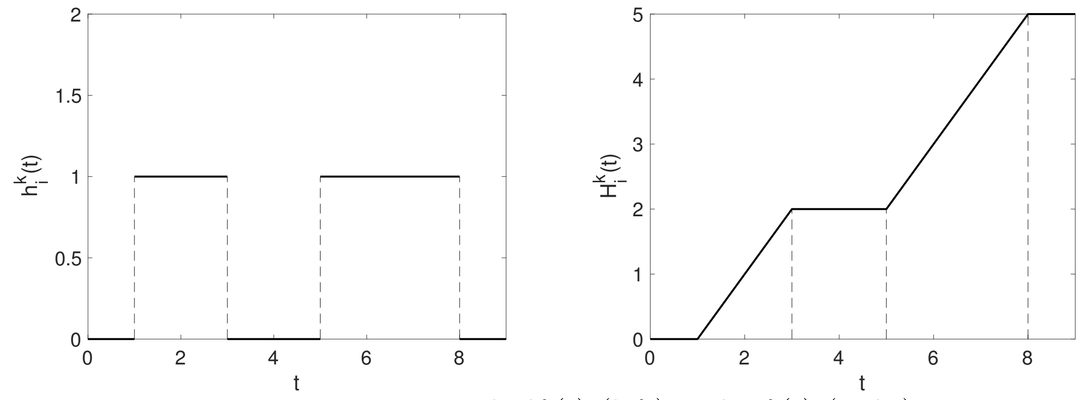


FIGURE 3.1: Example $h_i^k(t)$ (left) and $H_i^k(t)$ (right).

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).

3.3.2 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 has to examine every object (and hence runs in time linear in n), while the method of interval index has to examine 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. 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, which leads us to a theoretically optimal data structure proposed in [3] that can be used to answer a durable top-k query in time polylogarithmic in $|\mathcal{I}^k|$ and only linear in the number of result objects.

In the 3d halfspace reporting problem, we want to preprocess a set of points in $\mathbb{R}^3$ in a data structure such that all points below/above a given query plane can be reported efficiently. By duality [33], 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 $[a, b) \times [a', b') \subseteq \mathbb{T}^2$, 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 3.2. 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 $c = \mathsf{cnt}_i^k(a, a')$):

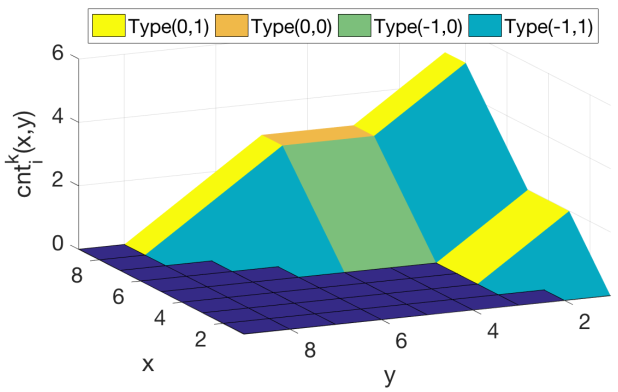


FIGURE 3.2: A geometric representation of $\mathsf{cnt}_i^k(x, y)$.

	$[a', b')$ 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 3.2 shows, $\mathsf{cnt}_i^k(x, y)$ consists of $O(|\mathcal{I}_i^k|^2)$ 3d 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 $\mathsf{DurTop}k([x, y), \tau) = \{i \in [1, n] \mid \mathsf{cnt}_i^k(x, y) \geqslant \tau \cdot (y - x)\}$. 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 [5], we can support this query in $O(N \operatorname{polylog} N)$ space and $O(\operatorname{polylog} N + |A|)$ 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 $O(\operatorname{polylog} N) = O(\operatorname{polylog}|\mathcal{I}^k|^2) = O(\operatorname{polylog}|\mathcal{I}^k|)$.

A Practical R-tree Implementation. As practical alternative to the theoretically optimal data structure from [5], 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 3.3 illustrates this transformation.

Shear transformations for other types can be similarly defined. We summarize

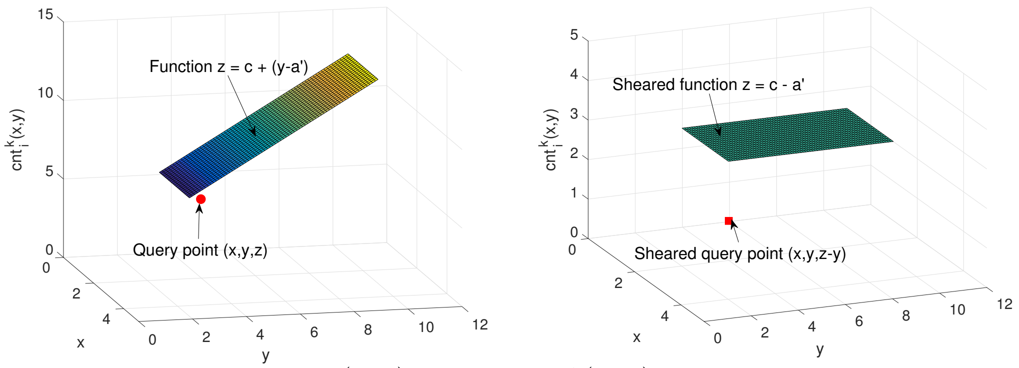


FIGURE 3.3: Shearing Type-$(0, 1)$ pieces of $\mathsf{cnt}_i^k(x, y)$ to be axis-aligned. The original coordinate space is on left and the transformed space on the right.

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.

3.4 Durable Top-k Queries 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 3.3. However, doing so is infeasible when there exist many possibilities for k. As discussed in Section 3.2, 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.

3.4.1 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 $\mathsf{dur}_i^k(I_R) = (\sum_{t \in I_R} \mathbf{1}[\mathsf{rank}_i(t) \leqslant$

$k])/|I_R|$. 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 $\mathsf{dur}_i^k(I_R) \geqslant \tau$, 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 :

Lemma 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, with probability at least $1-\delta$, $|\mathsf{dur}_i^k(I) - \mathsf{dur}_i^k(I_R)| \leqslant \epsilon\tau$.*

Proof. The key to the proof is the Chernoff-Hoeffding bound [56]. Fix an object i, and let X_t be a random variable indicating whether $t \in I$ is chosen in I_R; i.e., $X_t = 1$ if $t \in I_R$ or 0 otherwise. Let $F = \sum_{t\in I} X_t h_i^k(t) = \sum_{t\in I_R} h_i^k(t) = \mathsf{dur}_i^k(I_R) \cdot |I_R|$. Note that $\mathbf{E}[F] = \sum_{t\in I} \mathbf{E}[X_i] \cdot h_i^k(t) = \frac{|I_R|}{|I|} \cdot \sum_{t\in I} h_i^k(t) = \mathsf{dur}_i^k(I) \cdot |I_R|$. Therefore,

$$\mathbf{Pr}\big[|\mathsf{dur}_i^k(I) - \mathsf{dur}_i^k(I_R)| \geqslant \epsilon\tau\big] = \mathbf{Pr}\big[|F - \mathbf{E}[F]\,| \geqslant \epsilon\tau|I_R|\big].$$

Applying the Chernoff-Hoeffding bound, we have

$$\mathbf{Pr}\big[|\mathsf{dur}_i^k(I) - \mathsf{dur}_i^k(I_R)| \geqslant \epsilon\tau\big] \leqslant 2e^{-2|I_R|(\epsilon\tau)^2}.$$

This is the single bound for object i. To make this condition hold for each object we have seen in top-k over time, we also need apply the Union Bound. By setting $2e^{-2|I_R|(\epsilon\tau)^2} \leqslant \delta/(\frac{k}{\tau})$, where $\frac{k}{\tau}$ is the maximum possible size of τ-durable top-k objects over any query interval. Solving for $|I_R|$, we get $|I_R| = \frac{1}{2(\epsilon\tau)^2}\ln(\frac{2k}{\delta\tau})$. □

The lemma guarantees that with sufficient samples, this method with return any object with $\mathsf{dur}_i^k(I) \geqslant (1+\epsilon)\tau$ 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 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).

3.4.2 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 3.3.1, 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 $\mathsf{dur}_i^k(I) \geqslant \tau$, we would check whether $\mathsf{dur}_i^{k'}(I) \geqslant \tau$ 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.

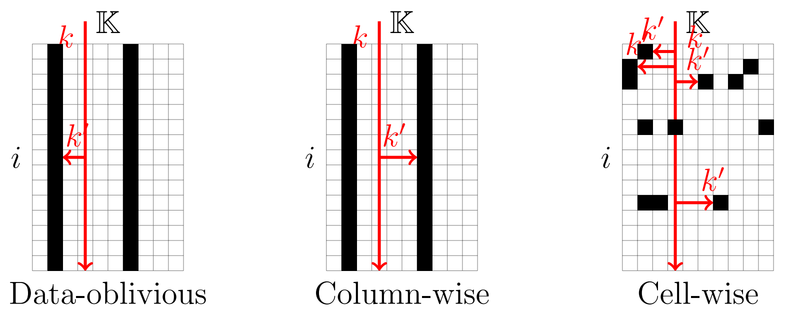


FIGURE 3.4: Illustration of three index-based methods.

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:

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

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 3.4. 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.[1] Unlike

[1] 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.

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 $n^\star \leqslant n$ 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 3.4, our index-based approach allows the storage budget to be set as an optimization constraint. Our experiments 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.

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 $\sum_{k\in K}\sum_{i=1}^{n}|\mathcal{J}_i^k| = \sum_{k\in K}|\mathcal{J}^k|$. 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 3.4.2 below will approach this problem in a more principled manner.

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 $\mathcal{Q}$ (multivariate in k, I, and τ) describing the query workload. For 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 $\mathbb{I} = \mathsf{supp}(\mathcal{Q}_I) \subseteq \{[a,b) \in \mathbb{T}\times\mathbb{T} \mid a \leqslant b\}$ 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

$$\underset{M,\mathsf{map}}{\text{maximize}} \qquad n - \mathbf{E}\big[A^k(I,\tau) \ominus \mathsf{DurTop}k(I,\tau)\big] \tag{3.1}$$

$$\text{subject to} \qquad \omega(M) \leqslant B. \tag{3.2}$$

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 $\delta_i(k,k';I,\tau)$ denote 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:

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

$$\text{where } \tau_1 = \mathsf{dur}_i^k(I),\ \tau_2 = \mathsf{dur}_i^{k'}(I),$$

$$\text{and } \Gamma_i(k,k';I) = (\min\{\tau_1,\tau_2\}, \max\{\tau_1,\tau_2\}].$$

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

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

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 $D_i(k) = \mathbf{Pr}[\tau \leqslant \mathsf{dur}_i^k(I)]$. Then $D_i(k)$ is non-decreasing in k, and $d_i(k, k') = |D_i(k') - D_i(k)|$.*

Proof. The non-decreasing nature of $D_i(k)$ follows directly from the fact that $\mathsf{dur}_i^k(I) \leqslant \mathsf{dur}_i^{k'}(I)$ for any $k' \geqslant k$—if object i is among the top k at any time t, then it must be among the top $k' \geqslant k$ at t as well.

Let $\tau_1 = \mathsf{dur}_i^k(I)$ and $\tau_2 = \mathsf{dur}_i^{k'}(I)$. Then

$$D_i(k) = \mathbf{Pr}[\tau \leqslant \tau_1] = \sum_{I \in \mathbb{I}} \sum_{\tau \leqslant \tau_1} p(\tau, I),$$

$$D_i(k') = \mathbf{Pr}[\tau \leqslant \tau_2] = \sum_{I \in \mathbb{I}} \sum_{\tau \leqslant \tau_2} p(\tau, I).$$

Noting that k is independent from I and τ, we clearly have

$$\begin{aligned} |D_i(k') - D_i(k)| &= \sum_{I \in \mathbb{I}} \sum_{\tau \in (\min\{\tau_1, \tau_2\}, \max\{\tau_1, \tau_2\}]} p(\tau, I) \\ &= d_i(k, k'). \end{aligned}$$

□

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 $\tau \leqslant \mathsf{dur}_i^k(I)$ 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 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 (3.1), we have

$$\begin{aligned} & n - \mathbf{E}\big[A^k(I,\tau) \ominus \mathsf{DurTop}k(I,\tau)\big] \\ & = n - \sum_{k \in \mathbb{K}} \left(p(k) \cdot \sum_{i=1}^{n} d_i(k, \mathsf{map}(i,k)) \right). \qquad (3.3) \end{aligned}$$

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

$$\underset{K}{\text{maximize}} \qquad n - \sum_{k \in \mathbb{K}} \Big(p(k) \cdot \min_{k' \in K} d(k, k') \Big) \tag{3.4}$$

$$\text{subject to} \qquad |K| \leqslant B. \tag{3.5}$$

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 $d(k, k') = |D(k') - D(k)|$.*

Proof. Assume $k' \geqslant k$ (the case of $k \leqslant k$ is analogous). By Lemma 2, $d_i(k, k') = D_i(k') - D_i(k)$ (we can remove $|\cdot|$ as D_i is non-decreasing). Hence, $0 \leqslant d(k, k') = \sum_{i=1}^{n} d_i(k, k') = \sum_{i=1}^{n} D_i(k') - \sum_{i=1}^{n} D_i(k) = D(k') - D(k)$. □

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 (3.4)–(3.5) for the column-wise method in has optimal substructure, as the following lemma shows.

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[2] for (3.4)–(3.5) with $\mathbb{K} = [k_1, k_2]$ and $B = b$. Then*

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

Proof. By Lemma 3, $\mathbb{K}$ can be embedded on a line. Choosing $K \subseteq \mathbb{K}$ would partition $\mathbb{K}$ into consecutive intervals, one for each $k^\star \in K$, such that for all k in this interval

[2] 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.

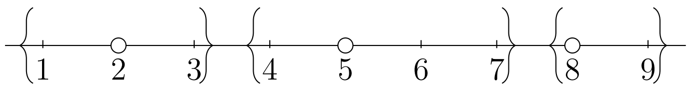


FIGURE 3.5: Partitioning of $\mathbb{K}$ by a chosen subset K. Each $k^\star \in K$ is shown as a circled point, and the interval of $\mathbb{K}$ that $k^\star$ is enclosed by { and }.

(which we shall refer to as the one covered by $k^\star$), $\arg\min_{k' \in K} d(k, k') = k^\star$, as illustrated in Figure 3.5.

Consider $K = \mathsf{OPT}([k_1, k_2], b)$, and specifically, the largest element $k^\star$ of K and the interval $[k, k_2]$ that $k^\star$ it covers. Clearly $k^\star$ must be chosen by $\mathsf{OPT}([k, k_2], 1)$, or else replacing $k^\star$ by $\mathsf{OPT}([k, k_2], 1)$'s choice would yield a better solution. Similarly, it is straightforward to show that $K \backslash \{k^\star\}$ must be chosen by $\mathsf{OPT}([k_1, k-1], b-1)$. $\square$

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.

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 $\omega(M) = \sum_{(i,k)\in M} |\mathcal{I}_i^k|$, since storing the prefix sums for entry (i, k) takes $|\mathcal{I}_i^k|$ space (index storage for supporting map is negligible in comparison). The optimization problem now becomes to

$$\underset{M}{\text{maximize}} \qquad n - \sum_{i}^{n} \left(\sum_{k \in \mathbb{K}} p(k) \cdot \min_{k' \in K_i} d_i(k, k') \right) \tag{3.6}$$

$$\text{subject to} \qquad \omega(M) = \sum_{(i,k)\in M} |\mathcal{I}_i^k| \leqslant B. \tag{3.7}$$

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

Lemma 5. *The optimization problem in* (3.6)–(3.7) *for the cell-wise method is NP-hard.*

Proof. First, recall the knapsack problem. Given a set of items $1, 2, \ldots, n$, each associated with a non-negative weight w_i and a non-negative value u_i, and a budget B, the knapsack problem finds a set of items $Q \subseteq [1, n]$ that maximizes $\sum_{i\in Q} u_i$ subject to $\sum_{i\in Q} w_i \leqslant B$.

We show that the knapsack problem can be reduced to the optimization problem for the cell-wise method. Consider an instance of the cell-wise indexing problem on n time series where $\mathbb{T} = [1, C_1 \cdot \sum_{i=1}^{n} u_i + C_2]$ (where C_1, C_2 are non-negative constant for scaling), I is fixed to be $\mathbb{T}$, k is fixed to be 1, and τ is drawn uniformly from $[0, 1]$. In other words, we only ask for τ-durable top-1 objects over the entire time domain. If object i is selected (i.e., $K_i = \{1\}$), then $d_i(1, 1) = 0$; otherwise ($K_i = \varnothing$), $d_i(1, \perp) = D_i(1) = \int_0^{\tau_i} p(\tau)\, d\tau = \tau_i$, where $\tau_i = \mathsf{dur}_i^1(\mathbb{T})$.

We carefully construct n time series such that their top-1 memberships over $\mathbb{T}$ satisfy the following two constraints: 1) for each object i, $\mathsf{dur}_i^1(\mathbb{T}) = \frac{u_i}{|\mathbb{T}|}$; 2) for each object i, $|\mathcal{I}_i^1| = C_2 \cdot \frac{w_i}{\sum_{i=1}^{n} w_i}$.

The construction works as follows. For each object i, we allocate $C_1 u_i$ time instants for it to be in the top-1, and we ensure that these time instants consist of exactly $C_2 \cdot \frac{w_i}{\sum_{i=1}^{n} w_i}$ non-adjoining intervals. It is not difficult to see that one can always partition $\mathbb{T}$ to achieve this construction by introducing a dummy object. More specifically, for each object i, we insert the dummy object $C_2 \cdot \frac{w_i}{\sum_{i=1}^{n} w_i} - 1$ times to break i's single interval into $|\mathcal{I}_i^1| = C_2 \cdot \frac{w_i}{\sum_{i=1}^{n} w_i}$ pieces. In total, we need insert the dummy object into the timeline as breakpoints C_2 times (including boundary positions), i.e., $|\mathcal{I}_{\text{dummy}}^1| = C_2$

With the data above, the knapsack problem is equivalent to

$$\begin{aligned} \underset{M \subseteq [1,n] \times \{1\}}{\text{maximize}} \quad & n - \sum_{i}^{n} \left(\min_{k' \in K_i} d_i(k, k') \right) \\ \text{subject to} \quad & \omega(M) \leqslant C_2 \cdot \frac{B}{\sum_i^n w_i}. \end{aligned}$$

In a non-trivial case that budget $B \leqslant \sum_i^n w_i$, any set that contains this dummy object is not feasible. Hence, the existence of the dummy object has no impact on our optimization problem. Intuitively, if an item i is selected by the knapsack problem, the objective function will gain u_i and pay cost w_i. On the other hand, if an object i is selected to be indexed, it will decrease the expected error by $\frac{u_i}{|\mathbb{T}|}$ and consume $C_2 \cdot \frac{w_i}{\sum_i^n w_i}$ space. Since the knapsack problem is NP-hard, so is the optimization problem for the cell-wise method. $\square$

Although the problem is NP-hard, the following lemma shows that its objective function is monotone and submodular [95].

Lemma 6. *The following function,*

$$\mathcal{G}(M) = n - \sum_i^n \left(\sum_{k \in \mathbb{K}} p(k) \cdot \min_{k' \in K_i} d_i(k, k') \right) \tag{3.8}$$

(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 $\theta = (i,k) \in ([1,n] \times \mathbb{K} \backslash) M_2$, we have:

$$\mathcal{G}(M_1) \leqslant \mathcal{G}(M_2), \text{ and} \tag{3.9}$$

$$\mathcal{G}(M_1 \cup \{\theta\}) - \mathcal{G}(M_1) \geqslant \mathcal{G}(M_2 \cup \{\theta\}) - \mathcal{G}(M_2). \tag{3.10}$$

Proof. $\mathcal{G}$ can be expressed as the sum of n functions: $\mathcal{G}(M) = n - \sum_{i=1}^n \mathcal{F}_i(K_i)$, where $\mathcal{F}_i(K_i) = 1 - (\sum_{k \in \mathbb{K}} p(k) \cdot \min_{k' \in K_i} d_i(k,k'))$. The class of submodular functions is closed under non-negative linear combinations. Hence, to prove $\mathcal{G}$ is submodular, it suffices to prove $\mathcal{F}_i$ is submodular.

Given object i, for any $k^\star \in K_i$, we define $\Phi_i(k^\star \mid K_i) = \{k \in \mathbb{K} \mid \arg\min_{k' \in K_i} d_i(k,k') = k^\star\}$ as the interval covered by $k^\star$. Consider adding $k_1, k_2 \in \mathbb{K} \backslash K_i$ to K_i. Let $S_1 = \Phi_i(k_1 \mid K_i \cup \{k_1\})$ and $S_2 = \Phi_i(k_2 \mid K_i \cup \{k_2\})$. First of all, it is clear that for all $k \in \mathbb{K} \backslash (S_1 \cup S_2)$, $\min_{k' \in K_i} d_i(k,k') = \min_{k' \in K_i \cup \{k_1\}} d_i(k,k') = \min_{k' \in K_i \cup \{k_2\}} d_i(k,k') = \min_{k' \in K_i \cup \{k_1,k_2\}} d_i(k,k')$; i.e., adding either k_1 and k_2 or both to K_i would not change the expected error incurred for such k's.

For $k \in S_1 \cup S_2$, we have

$$\min_{m=1,2} \{ \min_{k' \in K_i \cup \{k_m\}} d_i(k,k') \} = \min_{k' \in K_i \cup \{k_1,k_2\}} d_i(k,k'),$$

$$\max_{m=1,2} \{ \min_{k' \in K_i \cup \{k_m\}} d_i(k,k') \} \leqslant \min_{k' \in K_i} d_i(k,k').$$

Adding up the equality and inquality above, we get

$$\min_{k' \in K_i \cup \{k_1\}} d_i(k,k') + \min_{k' \in K_i \cup \{k_2\}} d_i(k,k')$$
$$\leqslant \min_{k' \in K_i \cup \{k_1,k_2\}} d_i(k,k') + \min_{k' \in K_i} d_i(k,k').$$

Summing the inequality above for all $k \in \mathbb{K}$, weighted by probability $p(k)$, and negating the inequality, we get

$$\mathcal{F}_i(K_i \cup \{k_1\}) + \mathcal{F}_i(K_i \cup \{k_2\}) \geqslant \mathcal{F}_i(K_i \cup \{k_1, k_2\}) + \mathcal{F}_i(K_i).$$

Therefore, $\mathcal{F}_i$'s and $\mathcal{G}$ are submodular. The proof for $\mathcal{G}$'s monotonicity is trivial, and hence omitted. □

It was shown in [83] 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 [102] that a modification of the greedy algorithm for solving the problem in [83] 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 7. *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*

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

Algorithm 1: Two-phase greedy algorithm.

```
Input  : Objective function $\mathcal{G}$ to maximize, additive cost function $\omega$,
         budget $B$, and candidate set $U = [1, n] \times \mathbb{K}$
Output: A subset $M^\star \subseteq U$ with $\omega(M^\star) \leqslant B$
1  $S_1 \leftarrow \varnothing$; $\max_1 \leftarrow 0$;
2  foreach $M \subseteq U$ where $|M| = 1$ or $|M| = 2$ do
3      if $\omega(M) \leqslant B$ then continue;
4      if $\mathcal{G}(M) > \max_1$ then
5          $\max_1 \leftarrow \mathcal{G}(M)$;
6          $S_1 \leftarrow M$;
7  $S_2 \leftarrow \varnothing$; $\max_2 \leftarrow 0$;
8  foreach $M \subseteq U$ where $|M| = 3$ do
9      if $\omega(M) > B$ then continue;
10     $S \leftarrow M$, $I \leftarrow U \backslash S$;
11     while $I \neq \varnothing$ do
12         $\theta \leftarrow \max_{\theta \in I} \frac{\mathcal{G}(S \cup \{\theta\}) - \mathcal{G}(S)}{\omega(\{\theta\})}$;
13         if $\omega(S \cup \{\theta\}) \leqslant B$ then $S \leftarrow S \cup \{\theta\}$;
14         $I \leftarrow I \backslash \{\theta\}$;
15     if $\mathcal{G}(S) > \max_2$ then
16         $\max_2 \leftarrow \mathcal{G}(S)$;
17         $S_2 \leftarrow S$;
18 if $\mathcal{G}(S_1) \geqslant \mathcal{G}(S_2)$ then return $S_1$;
19 else return $S_2$;
```

Proof. Follows directly from Lemma 6 and [102]. □

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 query accuracy, as experiments shows. We also optimize the procedure for finding the next greedy choice at each step using similar techniques in [111]. 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 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.

3.5 Coping with New Data

We now briefly discuss how methods in this section handle arrival of new data at the end of the time series. Here we focus on the more general case of variable k. The sampling-based method (Section 3.4.1) requires no any special handling of new data as it does not rely on indexing. The index-based methods (Section 3.4.2), on the other hand, must update the index structures as data arrives. For data-oblivious indexing (Section 3.4.2), since its choice of K does not depend on data, we simply need to append to the prefix sum H_i^k for each object i and each $k \in K$, based on the new ranking of objects. However, for data-driven indexing (Section 3.4.2), which decides what to index based on data characteristics, it is possible for old choices to become suboptimal with new data arrival. Re-optimizing and rebuilding the index with every new database state would be infeasible; hence, a different approach is needed.

A nice property of durable top-k queries is its decomposability over time. To answer a query over a time interval I, we can partition I into two sub-intervals I_1 and I_2, answer the query on I_1 and I_2 separately, and carefully combine the answers. The following lemma provides the foundation for this approach:

Lemma 8. *Consider two disjoint time intervals $I_1, I_2 \subseteq \mathbb{T}$. Then, for any k and object i,*

$$\mathsf{dur}_i^k(I_1 \cup I_2) \leqslant \max\{\mathsf{dur}_i^k(I_1), \mathsf{dur}_i^k(I_2)\}.$$

Proof. Let b_1 (b_2) denote the length of I_1 (I_2), and let a_1 (a_2) denote the number of time instants in I_1 (I_2) when object i is among the top k. We have

$$\mathsf{dur}_i^k(I_1) = \frac{a_1}{b_1},\ \mathsf{dur}_i^k(I_2) = \frac{a_2}{b_2},\ \text{ and } \mathsf{dur}_i^k(I_1 \cup I_2) = \frac{a_1 + a_2}{b_1 + b_2}.$$

$$\mathsf{dur}_i^k(I_1) - \mathsf{dur}_i^k(I_1 \cup I_2) = \frac{a_1 b_2 - a_2 b_1}{b_1^2 + b_1 b_2},$$

$$\mathsf{dur}_i^k(I_2) - \mathsf{dur}_i^k(I_1 \cup I_2) = \frac{a_2 b_1 - a_1 b_2}{b_2^2 + b_1 b_2}.$$

From the two numerators above it is easy to see that at least one of $\mathsf{dur}_i^k(T_1) \geqslant \mathsf{dur}_i^k(I_1 \cup I_2)$ and $\mathsf{dur}_i^k(T_2) \geqslant \mathsf{dur}_i^k(I_1 \cup I_2)$ must hold. □

The above lemma implies that any object in the answer of a τ-durable top-k query over $I = I_1 \cup I_2$ (where $I_1 \cap I_2 = \varnothing$) must be in the answer of a τ-durable top-k query over I_1 or I_2 (or both).

This observation naturally leads to the following approach to handling new data, which leverage old index structures for queries while leaving them intact. Suppose we have just constructed a data-driven index for our time series database up to time t. As time progresses and new data arrives, instead of updating the existing index, we index the data after time t in a "staging area." Since this area is only for a relatively small amount of newly arrived data, we can afford to store the prefix sums starting after t for all possible (i, k) pairs. A query over an interval that either precedes t or follows t can be answered by either the old index or the staging area, respectively. A query over an interval I that spans t will be decomposed into two—one over $I \cap (-\infty, t]$ against the old index and the other over $I \cap (t, \infty)$ against the staging area. The union of results returned by the two subqueries will serve as our candidate result set—Lemma 8 guarantees that every true result object will be returned by at least one of the two subqueries (barring approximation error introduced by the old index). We then further verify each candidate result object, which may require

Table 3.1: 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

accessing data beyond the old index, but the number of objects with this requirement is naturally bounded by the result set size for the subquery on the staging area.

Once the staging area is full, we re-optimize and rebuild the data-driven approximate index over entire time domain, effectively "extending" the old index to the current time and clearing the staging area for future data. It is also possible to design more elaborate schemes that maintain a sequence of indexes over time with different space/accuracy trade-offs, but they are beyond this scope of this discussion.

3.6 Experiments

In this section, we comprehensively evaluate the performance of all our methods. Section 3.6.1 compares the efficiency of various exact algorithms when k is fixed. For general durable top-k queries when k is variable, Section 3.6.2 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 Figure 3.1 and described in detail below.

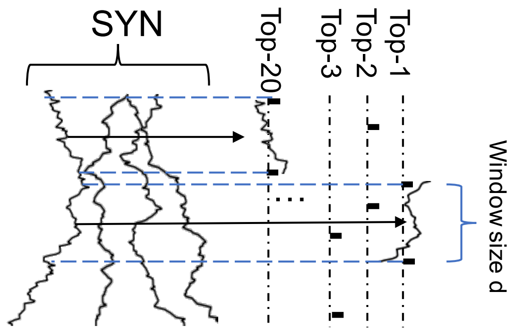


FIGURE 3.6: Generating *SynX* from *Syn*. Here, an object is boosted twice (at different times): once to top 20 and once to top 1.

Stock dataset contains daily transaction volumes of 3537 stocks in the United States from 2000 to 2009, collected by Wharton Research Data Services.[3] We treat each stock as a temporal object, and there are 2500 time instants.

Billboard dataset, obtained from the BILLBOARD 200 website,[4] 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 dataset, from the MesoWest project [57], 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 dataset refers to synthetic datasets with different sizes and distributions that we generate for in-depth comparison of various methods. Each time series is

[3] https://wrds-web.wharton.upenn.edu/wrds/

[4] http://www.billboard.com/charts/billboard-200

generated by an autoregressive model [106], 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 3.6. 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.

3.6.1 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 3.3.1. 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 3.3.2. TES is the state-of-the-art method from [108]. 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.

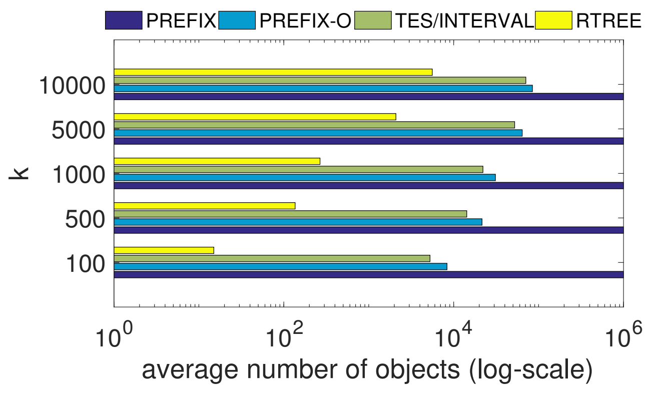
We present the results of our experiments on one large synthetic dataset. Here, we use Syn with one million objects, five thousand time instants, and $\sigma = 10$. Results reported in Figure 3.7 are obtained by averaging over 1000 random durable top-k queries with randomly drawn query intervals.

Figure 3.7a 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.

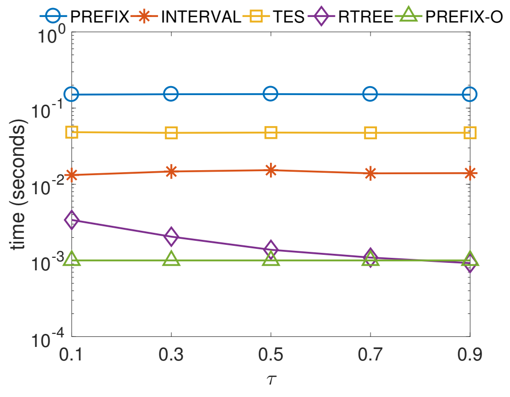
In terms of query time, however, the comparison is more nuanced. Figures 3.7b and 3.7c compare the query execution times of PREFIX, PREFIX-O, INTERVAL, TES, and RTREE, for two different settings of k, as we increase the durability threshold τ to make the queries more selective. First, in Figure 3.7b, 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 more selective queries as RTREE does. The basic version of PREFIX is the slowest here.

On the other hand, in Figure 3.7c, 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 3.7a), so the computational overhead of pruning and more complex index structures make them less attractive, even though they still examine fewer objects than PREFIX.

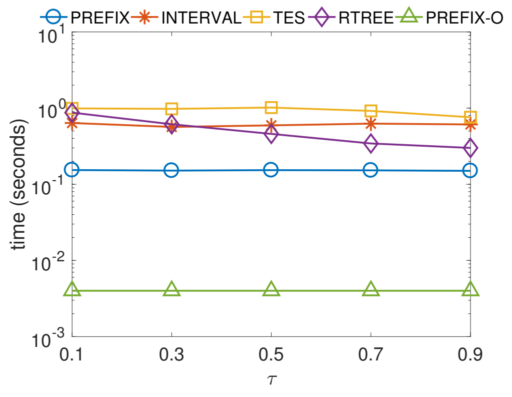
Overall, we conclude that PREFIX-O offers solid, competitive performance in practice, beating the theoretically more interesting RTREE except when queries are extremely selective. Because of its performance and simplicity, we also use PREFIX-O for our approximate index-based approach for handling the 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.



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



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

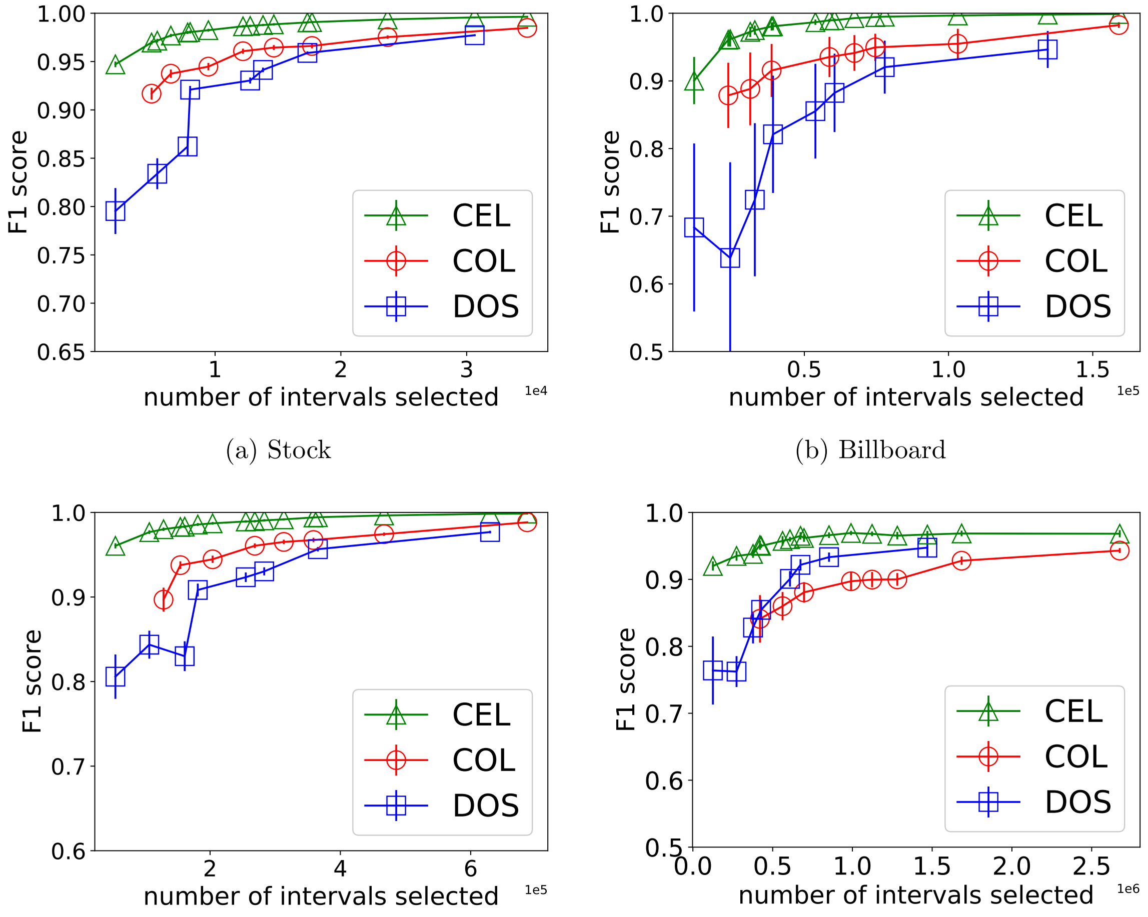


(c) Time per query; $k = 5000$

FIGURE 3.7: Comparing query efficiency for methods for the fixed-k setting. Dataset is *Syn*, with n = 1M, m = 5K, and $\sigma = 10$.

3.6.2 Variable-k setting

In this section, we continue to evaluate approximate methods for τ-durable top-k queries with variable k. Section 3.4 proposed two approaches for computing approximate answers: sample-based and index-based. We first evaluate the alternative methods for the index-based approach in terms of space and accuracy. Then, we compare 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, we evaluate the index construction costs of our index-based methods.

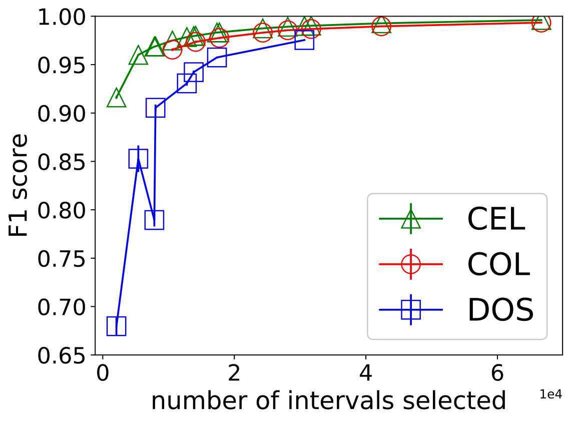


(a) Stock (b) Billboard

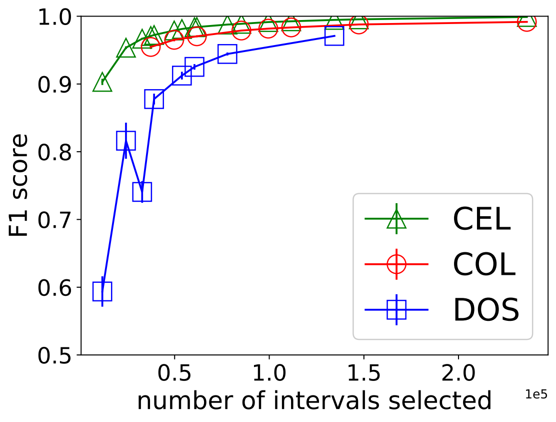
(c) Temp (d) Syn, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$

FIGURE 3.8: 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.

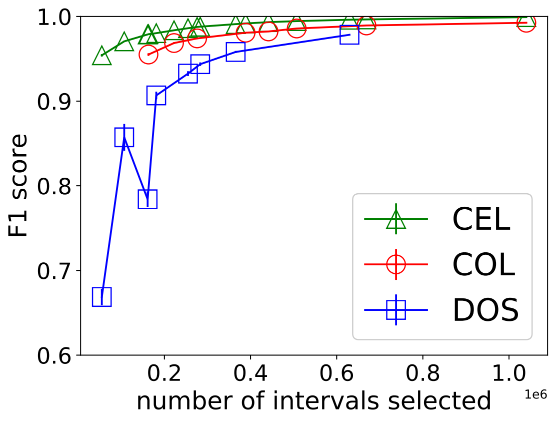
We use the standard F_1 score (harmonic mean of precision and recall) to measure the quality of approximate answers. The maximum possible F_1 is 1, achieved when both precision and recall are perfect. Since answer quality varies across query parameter settings, we experiment with various query workloads 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



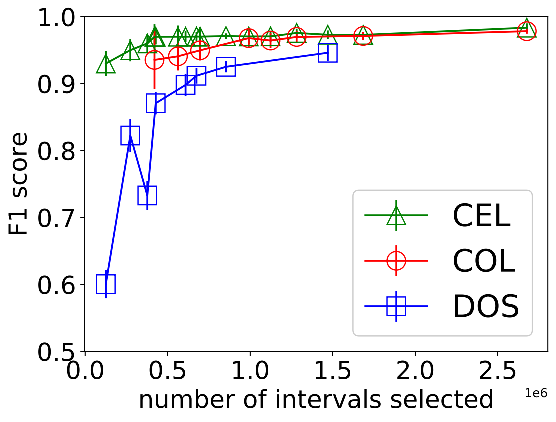
(a) Stock



(b) Billboard



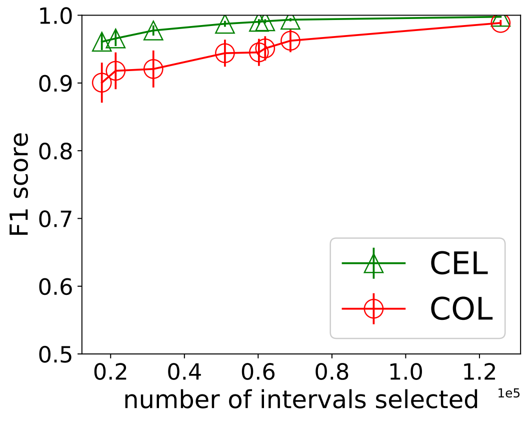
(c) Temp



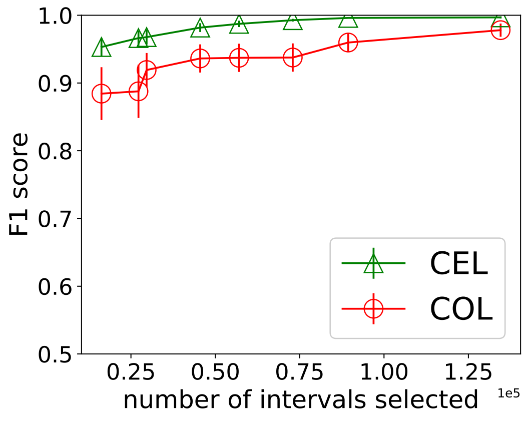
(d) Syn, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$

FIGURE 3.9: Quality of approximate answers by various index-based methods; $k \sim \mathcal{N}(50, 15^2)$ and $\mathbb{K} = [1, 500]$; uniform I.

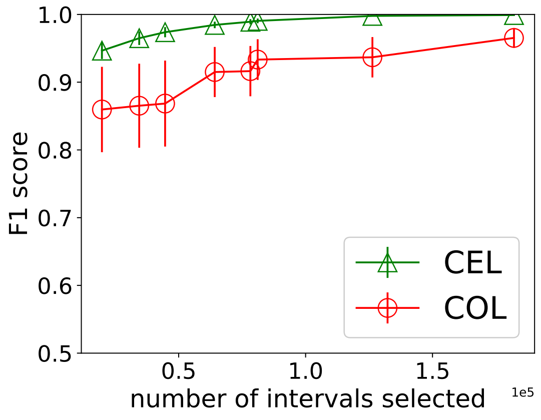
likely query with high τ and small k, but they may still try larger k or lower τ more often than a normal distribution would suggest. We typically draw the endpoints of I uniformly at random, sometimes with interval length restricted to appropriate ranges. Additional details will be given with the experiments. 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.



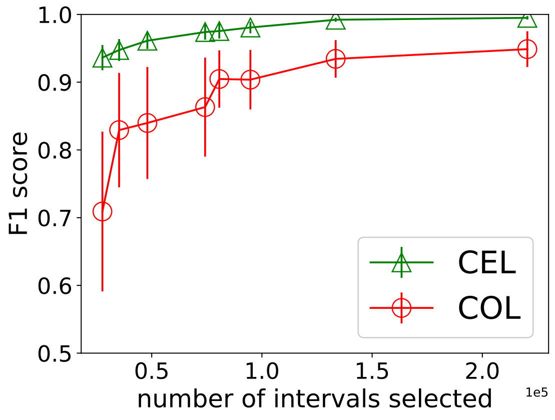
(a) $d = 100$



(b) $d = 50$



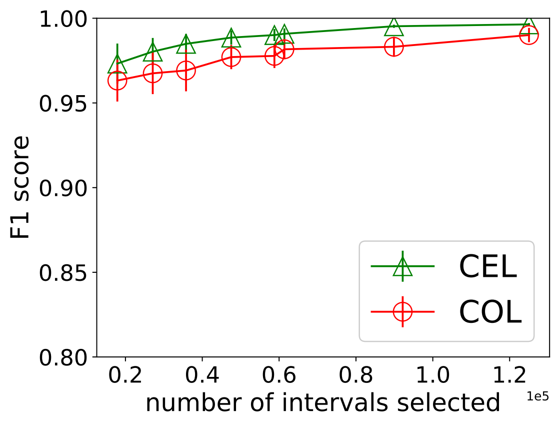
(c) $d = 20$



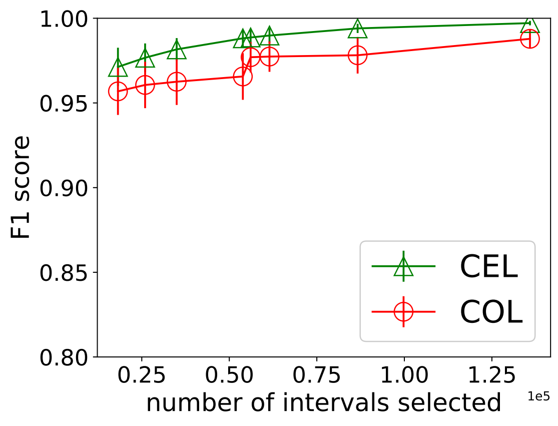
(d) $d = 10$

FIGURE 3.10: Quality of approximate answers by CEL vs. COL on *SynX* with $n = 1\text{K}$, $m = 1\text{K}$, $\sigma = 20$; $\ln(k) \sim \mathcal{N}(3, 0.5^2)$ and $\mathbb{K} = [1, 500]$; uniform I.

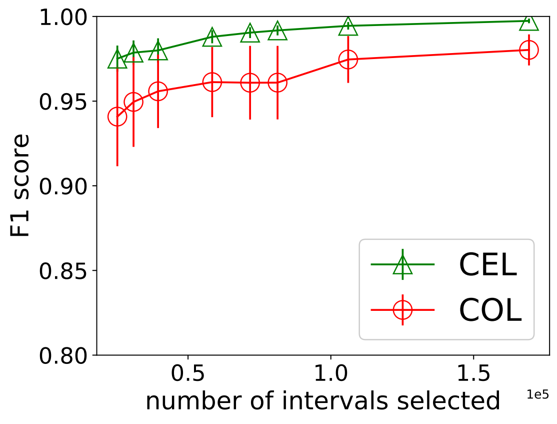
Approximate Index-Based Methods. Here we compare the three index-based methods we proposed in Section 3.4.2: data-oblivious indexing (DOS), column-wise indexing (COL), and cell-wise indexing (CEL). Note that all these methods allow the index size to be adjusted, which affects their approximation quality. 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



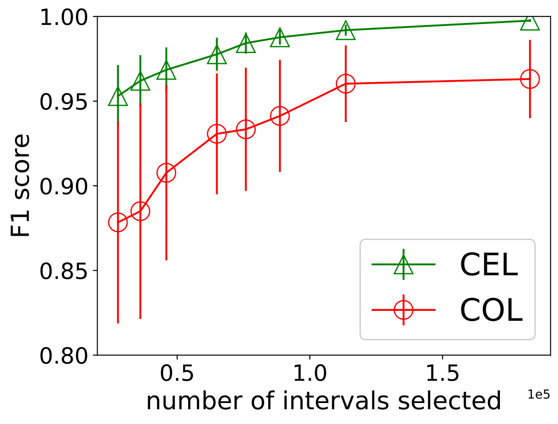
(a) $d = 100$



(b) $d = 50$



(c) $d = 20$



(d) $d = 10$

FIGURE 3.11: Quality of approximate answers by CEL vs. COL on *SynX* with $n = 1\text{K}$, $m = 1\text{K}$, $\sigma = 10$; $k \sim \mathcal{N}(50, 15^2)$ and $\mathbb{K} = [1, 500]$; uniform I.

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 3.8 and 3.9 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 3.8, but a normal distribution in Figure 3.9; 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.[5] 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, we see that DOS is also the worst among the three methods; CEL again is the best, consistently delivering high accuracy with very little variation among individual queries. Finally, between Figures 3.8 and 3.9, 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 3.8 is "harder" than the normal distribution used in Figure 3.9, 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.[6] 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).

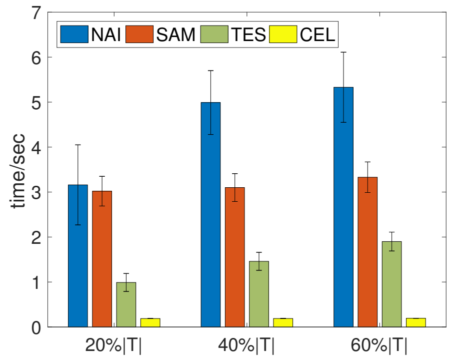
[5] This behavior also explains why in Figure 3.8d, COL does seemingly worse than DOS: given the same number of columns to index, COL in fact does offer higher accuracy than DOS, but it also chooses columns that require more space, hence pushing its curve to the right of that of DOS.

[6] 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.

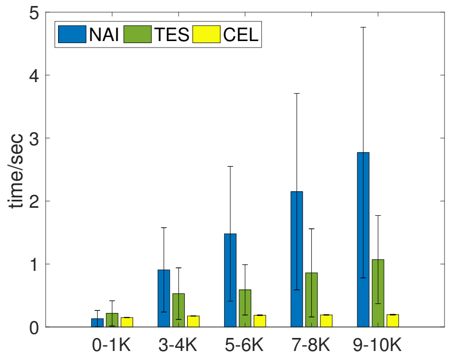
We use SynX with $\sigma = 20$ and vary d, where a smaller d leads to more frequent rank ranges. Figure 3.10 shows the results when k in the query workload follows a log-normal distribution (as in Figure 3.8). We focus on comparing just COL and CEL here because DOS is clearly inferior. In Figure 3.10, we see that, as d decreases and rank change complexity increases (from left to right), 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.

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.

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



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



(b) Varying $\mathbb{K}$; uniform I

FIGURE 3.12: Query execution times for various durable top-k solutions. *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$.

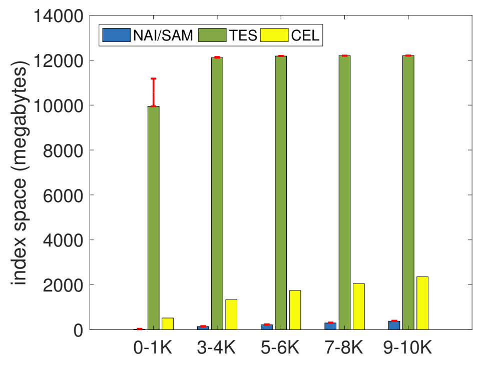


FIGURE 3.13: Index space for various durable top-k solutions. *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$; uniform I (relevant to only CEL).

solution, which precomputes and stores the top-$k_{\max}$ 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 3.4.1; it materializes exactly the same information as NAI. TES is our implementation of the state-of-the-art exact solution from [108]. Since its query performance depends on the actual data structures used, we take care to discount any possible dependency when taking measurements for TES.[7] 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.

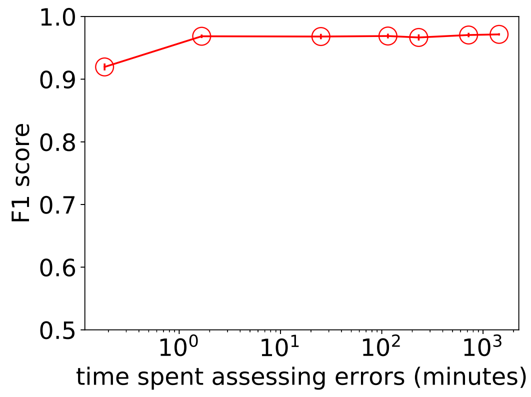
Figure 3.12a shows how the length of the query interval I influences query execution times of various approaches. Here, we draw I's starting point randomly, and make their lengths span 20%, 40%, or 60% of the entire time domain. We draw k from $\mathcal{N}(5500, 100^2)$, truncated to $[5000, 6000]$. 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.

Figure 3.12b 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

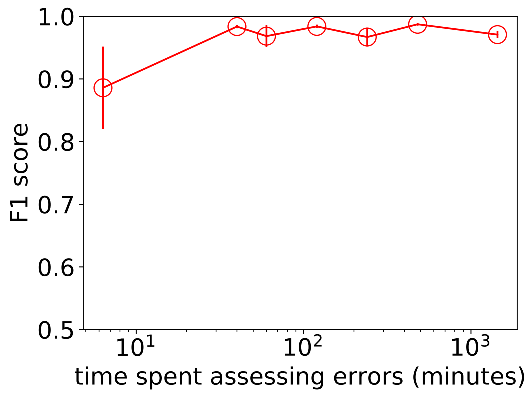
[7] TES uses a non-trivial data structure for reporting all rank changes within k during the query interval, and we do not have access to its original implementation. Hence, for the execution times of TES we report in these experiments, we simply exclude the time spent using our implementation of this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

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 3.12b, 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 3.13, as measured by the amounts of space consumed by Python data structures. The query workloads are the same as those in Figure 3.12b. 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 3.13, 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



(a) *Temp*



(b) *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$

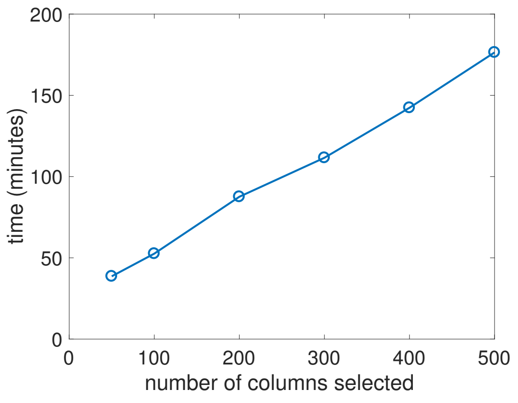
FIGURE 3.14: CEL index quality as a function of optimization time spent on assessing expected error using Monte Carlo simulations during optimization; same query workload as Figure 3.8.

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.

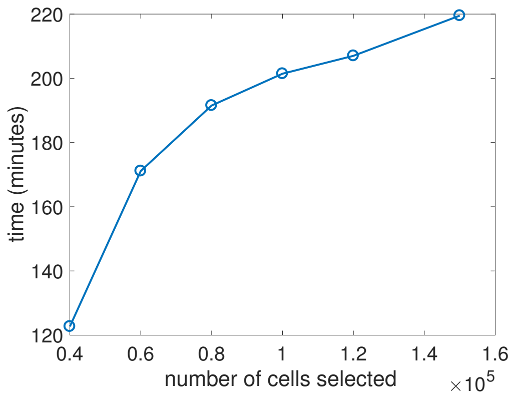
To further demonstrate scalability of CEL, we test it on an even larger version of Syn with 50 billion data points ($n = 10\text{M}$, $m = 5\text{K}$, and $\sigma = 10$). In the query workload, I is uniform and $\mathbb{K} = [5000, 6000]$. Under this setting, CEL only needs 3.6GB of index space to deliver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

Index Construction. The two data-driven index-based methods, COL and CEL, perform elaborate preprocessing and optimization during their index construc-



(a) Dynamic programming for COL



(b) Greedy for CEL

FIGURE 3.15: Optimization time as a function of budget. *Syn*, $n = 1\text{M}$, $m = 5\text{K}$, $\sigma = 10$, $\mathbb{K} = [9\text{K}, 10\text{K}]$

tion step. In this section, we evaluate the performance of index construction for these two methods and demonstrate their 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 3.4.2, 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 3.14 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 3.8. 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 3.14 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 3.14, 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 3.4.2) and greedy for CEL (Section 3.4.2). Figure 3.15 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 = 1M, m = 5K, σ = 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 3.8. As we can see in Figure 3.15, generally speaking, bigger space budgets result in longer optimization times, and CEL optimization is more expensive than COL optimization. At a moderate bud-

Table 3.2: 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

get settings for CEL, shown as the third data point in Figure 3.15(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.

Finally, Figure 3.2 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 3.8. For Syn with 5 billion data points (n = 1M, m = 5K, $\sigma = 10$), we use the query workload as in Figure 3.15. For Syn with 50 billion data points (n = 10M, m = 5K, $\sigma = 10$), we use the same query workload as the one used for this dataset in Section 3.6.2. As shown in Figure 3.2, 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.

3.7 Conclusion

In this chapter, 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.

4

Durable Top-k Queries on Instant-Stamped Temporal Data

SUCCESS is not about HERE and NOW. you must pass the TEST OF TIME.

Mark Batterson

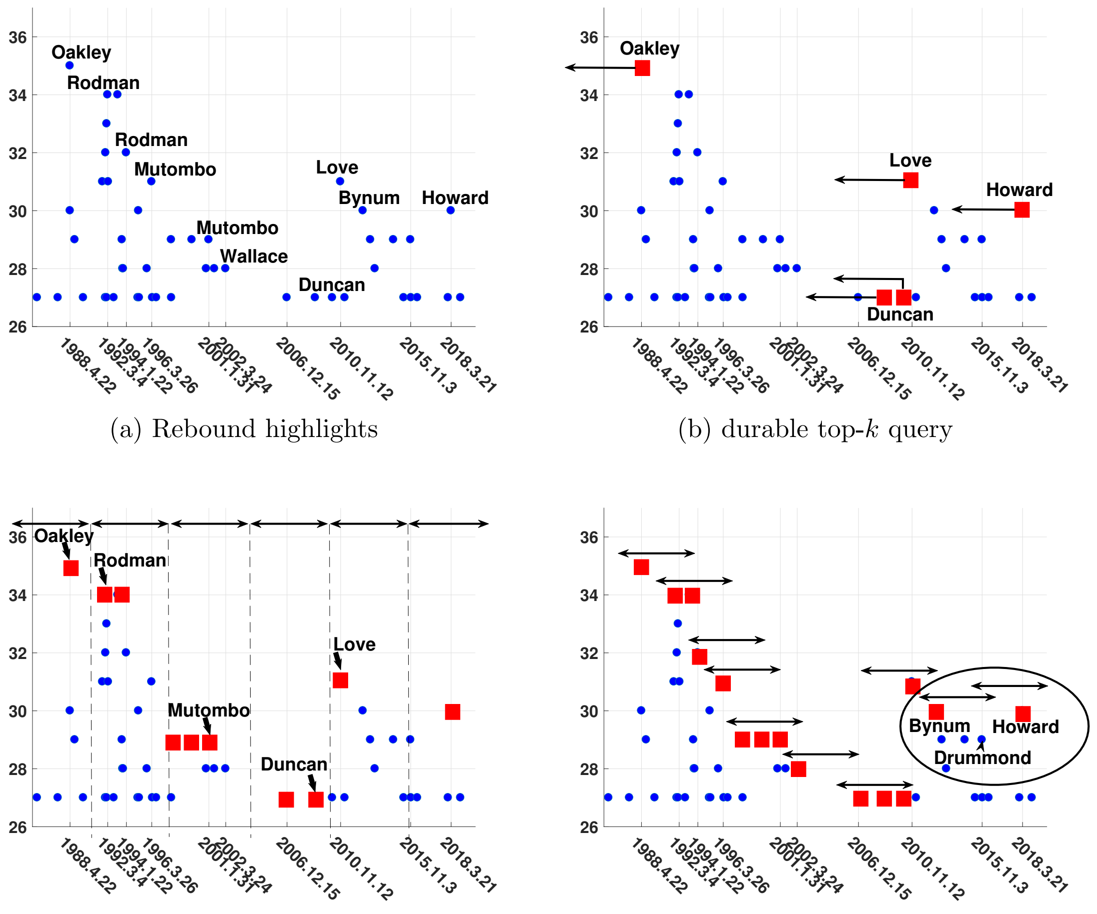
4.1 Introduction

In Chapter 3, we have introduced an interesting type of durability queries, namely "τ- durable top-k queries", on sequence-based temporal data. Such queries search for temporal object with consistently exceptional performances over time; i.e., ranking as top-k (in daily basis) for a large portion of times during a given time period. An novel indexing method and corresponding query processing technique were proposed to answer durability queries with low index space, fast query time and high quality answers.

In this Chapter, we continue to explore durability queries, generalizing from statement as in Example 2, on instant-stamped temporal data. Formally, instant-stamped temporal data consists of a sequence of individual records, each timestamped by a time instant which we call the arrival time, and ordered by the arrival time. Such data is ubiquitous in a rich variety of domains; i.e., sports statistics, weather measurement, network traffic logs and e-commerce transactions. Analysis of durability on instant-stamped temporal data is a useful part of the toolbox for anybody who works with historical data, and can be particularly helpful for journalists and marketers to identify newsworthy facts and communicate their impressiveness to the public.

In this chapter, we consider "τ-durable top-k queries" for finding instant-stamped records that stand out in comparison to others within a surrounding time window. Though sharing the same name as the durability query we introduced in Chapter 3, τ here represents the time duration of records' consistent dominance, as opposed to τ a percentage of times in previous work.

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



(a) Rebound highlights (b) durable top-k query

(c) Tumbling Window Top-k (d) Sliding Window Top-k

FIGURE 4.1: A case study on finding durable noteworthy rebound performances in NBA history. Red squares highlight results returned by different queries, and line segments represent the durability time window.

durability of its supremacy. For instance, the Kobe Bryant statement in Example 2 is based on one record returned from a durable top-k query where $k = 1$, $\tau = 45$ years (looking back) or $\tau = 15$ years (looking forward), and ranking is done by a simple f that just returns player's total points in a single NBA game.

Note that there are different ways for capturing the notion of durability in queries, including some types that have been studied in the past. To understand why we choose our definition of durable top-k queries, we examine the alternatives with a simple concrete example.

Example 4. Suppose we are interested in finding exceptional rebounds performances (by individual players in individual games) in NBA history—particularly, those that stood out as the top record (or tying for the top record) in a 5-year time span. Figure 4.1.1 plots all relevant records (i.e., no fewer than 27 rebounds by a single player in a single game) in entire NBA history. We consider the following three queries to accomplish our task; the latter two have been widely studied in the stream processing and top-k query processing literature. Note that in this example $k = 1$.

- Durable top-k (our query): This is the query that we propose. For each record, we look back in a 5-year window ending at the timestamp of the record, and check whether the record has the top score among all records within this window. Figure 4.1.2 highlights the records (red squares) returned by our query; for each result record, we also show its durability window (as a line segment), i.e., a 5-year window ending at the record for which it remains on the top.

- Tumbling-window top-k: This query first partitions the timeline into a series of non-overlapping, fixed-sized (5-year) windows, and then returns the top record within each time window. The placement of the windows is up to the user and can affect results. Results for one particular placement of the windows are shown in Figure 4.1.3.

- Sliding-window top-k: This query slides a 5-year window along the timeline, and returns the top record for each position of the sliding window. Figure 4.1.4 highlights a few representative sliding windows, as well as the top records during these windows.

All these queries are able to uncover some meaningful durable top records; i.e., for any data record (X, Y, Z) marked as a red square in Figure 4.1, we can claim "player X grabbed Y rebounds in a game on date Z, which is the best in some 5-year span."

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

However, there are some notable differences among the results found by these queries.

- Tumbling-window vs. our query: The general observation is that the results of tumbling-window are highly sensitive to the choice of window placement. In Figure 4.1.3, tumbling-window picks (Mutombo, 29, 2001) and the other two performances with 29 rebounds as they were the best ones during 2000-2005, but there were more impressive performances right before them, unfortunately leaving the impression that they stood out only because the windows were cherry-picked. Furthermore, if we choose to place all windows slightly to the right such that the last window ends with the most recent arrival time, (Rodman, 34, 1992) will be eliminated by (Oakley, 35, 1988), and (Duncan, 27, 2009) will be overlooked since it is shadowed by (Love, 31, 2010). Overall, because of high sensitivity to window boundaries, tumbling-window runs the risk of omitting important records as they happen to be overshadowed by some other records in the same window, and picking less interesting records as they happen to be the top ones in that specific window.

- Sliding-window vs. our query: Sliding-window is not susceptible to window placement, but it effectively considers all possible window placements, and it returns the union of all top records for each such placement. This approach leads to possibly many records that are not as meaningful in practice. In Figure 4.1.4, sliding-window apparently returns overwhelmingly more results compared to our query, which makes it less applicable to mining most noteworthy records. Even more unnatural is the fact that as we slide the window along the timeline, a record can come in and out of the result; i.e., there is no continuity. To illustrate, suppose we are interested in durable top-2 records with 5-year windows, and let us focus on Drummond's 29 rebounds performance on 2015.11.3 (highlighted in Figure 4.1.4). It is surrounded by two top performance (Howard, 30, 2018) and (Bynum, 30, 2013). Sliding-window will return this record when the window is positioned at 2014-2019, but not when positioned at 2013-2018; however, the record will be returned again when the window moves to 2012-2017. Such discontinuity makes the results rather unnatural to interpret.

In comparison, our query does not have the issue of sensitivity to window placement or that of difficulty of interpretation, because we assess each record in a 5-year window that leads up to its own timestamp. Thus, our query result records can be consistently interpreted as having durability "within the past 5 years" and clearly communicated to the audience. The results from the other two queries would have be qualified with rather specific durability windows, [1] which may be perceived as cherry-picking. In general, we argue that consistency and simplicity of our query makes it more applicable to journalists, marketers, and data enthusiasts alike who seek result that are easily explainable to the public.

Although the above example ranks records by just a single attribute, its argument

[1] A related question is whether we can post-process the results of the sliding-window query to obtain the results to our query; e.g., filtering those result records in Figure 4.1-(4) to get those in Figure 4.1-(2). Unfortunately, such an approach, which we consider as one of the baselines in our experiments, is prohibitively slow when dealing with large temporal datasets, as we shall show in later sections.

can be easily extended to the more general case where ranking is done by a user-specified scoring function that combines multiple attribute values into a single score.

Contributions. Our contributions are as follows:

- We propose to find "interesting" records from large instant-stamped temporal datasets using durable top-k queries. Compared with other query types related to durability, our query produce results that are more robust (i.e., less sensitive to window placement than tumbling-window) and more meaningful (i.e., easier to interpret than sliding-window).
- We propose a suite of solutions based on two approaches that process "promising" records in different prioritization order. We provide a comprehensive theoretical analysis on complexities of the problem and of our proposed solutions.
- Our solutions are general and flexible. They do not require any specific scoring function f, but instead assume a well-defined building block for answering top-k queries using f, which can be "plugged into" our solutions and analysis. We give some concrete example of f and the building block later in this section. In particular, f can be further parameterized according to user preference; these parameters, along with I, k, and τ, can be specified at query time, making our solutions flexible and suitable for scenarios where users may explore various parameter setting at run-time (either interactively or automatically).
- We show that the query time complexity of our best algorithms is proportional to $O(|S| + k\lceil \frac{|I|}{\tau} \rceil)$ in the worst case, where $|S|$ is the answer size. Furthermore, we prove that the expected answer size of a durable top-k query $|S|$ is $O(k\lceil \frac{|I|}{\tau} \rceil)$ under the random permutation model (where the data values can be arbitrarily distributed but arrival order is random); this result implies that our best algorithms are in a sense optimal because this complexity term is essentially linear in the output size.

- Extensive experiments on both real and synthetic data, with various combinations of query parameters and data dimensionalities, demonstrate the efficiency our methods over baseline solutions. Our best solutions can be up to 2 orders of magnitude faster in practice.

Chapter Overview. In a nutshell, our proposed algorithms 1) visit promising records in some manner, and 2) check the durability (with respect to a top-k query) for each record we visit. Techniques for improvement mostly focus on how to efficiently identify candidate records and eventually reduce the total number of durability checks in the second phase. Our proposed algorithms come in two flavors: time-prioritized and score-prioritized, introduced in Section 4.3 and Section 4.4, respectively. The time-prioritized solution traverses and finds candidate records sequentially along the timeline, while the score-prioritized solution greedily chooses unvisited candidates with the maximum score (with respect to f). Though in different manners, we show in later sections that these two solutions actually equivalently reduce and bound the size of candidate records (or, the number of durability checks). More interestingly, in Section 4.5, we further demonstrate that the bound is proportional to the answer size of a durable top-k query, which means our algorithms run faster when the query is more selective, e.g., with smaller k or longer durability τ. Section 4.6 experimentally evaluates all our proposed solutions. We review related work in Section 4.7 and conclude in Section 4.8.

4.2 Problem Statement and Preliminaries

Problem Statement. Consider a dataset P with n records, where each record $p \in P$ has d real-valued attributes and is represented as a point $(p.x_1, p.x_2, \ldots, p.x_d) \in \mathbb{R}^d$. For simplicity, we consider a discrete time domain of interest $\mathbb{T} = \{1, 2, \ldots, n\}$, and let $p.t \in \mathbb{T}$ denote the *arrival time* of p. All records in P are organized by increasing order of their arrival time. Given a non-empty time window $W : [t_1, t_2] \subseteq \mathbb{T}$, let $P(W)$ denote the set of records that arrive between t_1 and t_2; i.e., $P(W) = \{p \in P \mid t_1 \leqslant p.t \leqslant t_2\}$.

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

For simplicity of exposition, we consider durability windows ending at the arrival time of each record throughout the paper (i.e., the "looking-back" version), but our solution can be extended to the more general case where the windows are anchored consistently relative to the arrival times (including the "looking-ahead" version). We say a record p is τ-durable[2] if $p \in \pi_{\leqslant k}([p.t - \tau, p.t],)$. That is, p remains in the top-k for τ time during $[p.t - \tau, p.t]$. We are interested in finding records with long durability. Given a query interval I and a durability threshold $\tau \in [1, |\mathbb{T}|]$, a *durable top-k* query, denoted $\mathsf{DurTop}(k, I, \tau)$, returns the set of τ-durable records that arrive during I; i.e., $\mathsf{DurTop}(k, I, \tau) = \{p \in P(I) \mid p \in \pi_{\leqslant k}([p.t - \tau, p.t],\})$.

Scoring Function and Top-k Query Building Block. As discussed earlier, our proposed algorithms and complexity analyses are applicable to any user-specified scoring function f as long as there exists a "building block" that can answer basic (non-durable) top-k queries under f. This building block can be a "black box": the novelty and major contribution of our algorithms come from its ability to reduce and bound the number of invocations of the building block, totally independent of how the building block operates itself. Of course, the overall algorithm complexity still depends on the efficiency of the building block. For a function f, we consider that an

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

index of size $O(s(n))$ can be constructed in $O(u(n))$ time that answers top-k queries with respect to f in $O(q(n)+k)$ time, where n is the data size and $s(\cdot), u(\cdot), q(\cdot)$ are functions of n.

In this paper, we are more interested in top-k queries on a subset of data specified by a time window W given at query time; i,e., computing $Q(k, W)$ that reports the k records in $P(W)$ with the highest scores with respect to f. With a slight care, the top-k query building block can be used to solve this problem by paying a logarithmic factor in index size, query time and construction time. That is, for a function f we can construct an index of size $O(s(n)\log n)$ in $O(u(n)\log n)$ time so that for given k, W, $Q(k, W)$ can be computed in $O((q(n)+k)\log n)$ time.

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

- *linear*: $f(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$,
- *linear combination of monotone scoring functions*: $f(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot h(p.x_i)$, where h is a monotone function; i.e., $h(\cdot) = \log(\cdot)$,
- *cosine*: $f(p) = \frac{1}{|p||\mathbf{u}|} \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$,

where $\mathbf{u}$ is a real-valued preference vector and $f_{\mathbf{u}}$ denotes that the scoring function f is parameterized by $\mathbf{u}$. We refer to this class of functions as *preference functions*. Top-k queries using such class of scoring functions (preferably in the above three forms) have been well studied over the past decades both in computational geometry [3, 22, 80, 4, 9, 19] and databases [20, 116, 58, 61]. For example, for preference functions above, there is an index with $u(n) = O(n)$, $s(n) = O(n)$, and $q(n) = O(n^{1-1/\lfloor d/2 \rfloor})$, skipping polylog$(n)$ factors.

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

Sliding-Windows and Baseline Solution. Recall from the discussion in Example 1 (Figures 4.1-(2) and 4.1-(4)) that there is a connection between our problem and the sliding-window version, which has been well studied [82, 66, 32]. Indeed, one of our baseline solution is adopted from [82] with incremental top-k maintenance over sliding windows. However, the standard sliding-window technique is more suitable for data streams, where it is necessary to linearly scan incoming data. Instead, our query analyzes static historical data. The linear complexity of sliding windows becomes infeasible especially when dealing with large temporal datasets. The limitation hence motivates our solutions in later sections. Experimental results demonstrate our proposed algorithms' significant efficiency gain (up to 2 orders of magnitude) over sliding-window baselines.

4.3 Time-Prioritized Approach

The time-prioritized approach is straightforward: we visit records in time order and check their durability. We start with a baseline approach (Section 4.3.1) and propose an improved version (Section 4.3.2) using the observation that we can skip many unpromising records in practice. What is more interesting is how this simple improvement leads to provably substantial reduction in complexity (Section 4.3.3).

4.3.1 Time-Baseline Algorithm

We start with a baseline solution, referred to as Time-Baseline or T-Base. T-Base shares the same spirit as the solution proposed in [82], where authors studied the

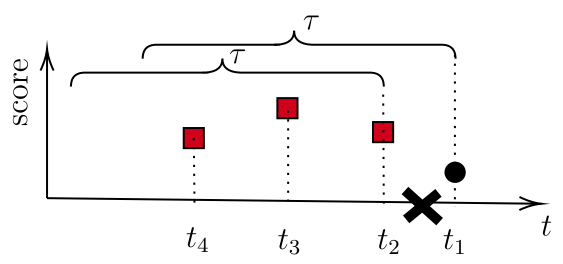


FIGURE 4.2: Data skipping in Time-Hop Algorithm.

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

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

4.3.2 Time-Hop Algorithm

It is not hard to see that the durable top-k query can be viewed as an offline version of the top-k query in the sliding-window streaming model. Hence, the baseline algorithm introduced above does not best serve our needs. Since the entire data is available in advance, the manner of continuous sliding window wastes too much time on those non-durable records. After all, a meaningful durable top-k query should be selective.

Before describing the algorithm, we illustrate the main idea using an example for $k = 3$, shown in Figure 4.2. By running a top-3 query $Q(3, [t_1 - \tau, t_1])$, consider the record p arriving at t_1 (black circle) is not τ-durable; i.e., $p \notin \pi_{\leqslant 3}([t_1 - \tau, t_1],)$. We know the current top-3 set contains records (red squares) that arrive at t_4, t_3 and t_2. Then, no records arriving between t_2 and t_1 would be τ-durable and we can safely hop from t_1 to t_2. This simple and useful observation simplifies the query procedure, and allows larger strides for sliding windows.

Now, we present our algorithm Time-Hop (T-Hop) (the pseudocode can be found in Algorithm 2). For each record we visit with timestamp t_i, we run a top-k query in $[t_i - \tau, t_i]$ (Line 4). If the record is not durable, we slide the window back to the most recent arrival time of records, say t_j, in the current top-k set (Line 9), skipping the non-durable records between t_j and t_i. Otherwise, if a durable record is found, we slide the window backwards by 1 (Line 7) as usual. Note that if we adopt the look-ahead version of durability, we just need to reverse the traversal order (and time-hopping) on timeline as well.

4.3.3 Complexity Analysis of T-Hop

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

Algorithm 2: T-Hop (k, I, τ)

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

Lemma 9. *The total number of top-k queries performed by the Time-Hop algorithm is $O\big(|S| + k\lceil \frac{|I|}{\tau} \rceil\big)$.*

Proof. We start with a high-level sketch of the full proof.

For each record we visit in T-Hop, a top-k query is called for a durability check. If the record is not τ-durable, we refer it to as a "false check". Otherwise, we add it to the answer set. Hence, we only need to bound the total number of false checks. We decompose the total number of false checks into a set of disjoint τ-length windows, and derive an upper bound of false checks that happen in such a window. In particular, let ρ be a window of length τ and let S_ρ be the τ-durable records in ρ. We divide the false checks in ρ into two types. If a false check appears immidiately after a τ-durable record (found by the algorithm) then this is a type-1 false check. Otherwise it is a type-2 false check. From the definition, the number of type-1 false checks in ρ is $O(S_\rho)$. Furthermore, we show that after finding i type-2 false checks in ρ, a top-k query (that is called for durability check) can only find $k - i$ records in ρ. In that way we show that the number of type-2 false checks is $O(k)$. Given a query interval I, there are at most $\lceil \frac{|I|}{\tau} \rceil$ disjoint τ-length sub-intervals. We conclude that the number of top-k queries is $O\big(|S| + k\lceil \frac{|I|}{\tau} \rceil\big)$.

Now we elaborate the full proof.

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

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

Recall that if the algorithm visits a point p it computes the top-k elements in $[p.t-\tau, p.t]$. Let U_p be the list of the top-k items in $[p.t-\tau, p.t]$. Let $Z_p = U_p \cap \rho$, be the list of these top-k elements that lie in ρ. Generally we refer to Z_p as a Z list. At the beginning of the algorithm assume that we find the top-k elements in ρ, so we have a list Z with $|Z| = k$. We show the following two observations. i) Each time that the algorithm finds a type-2 false check the new Z list of top-k points in ρ has cardinality at least one less than the previous list. ii) The cardinalities of the Z lists as we run the algorithm in ρ are never increasing. If we show (i), (ii) we could argue that after the algorithm finds k type-2 false checks in ρ, the Z list will be empty and the algorithm will visits a point out of ρ.

Let Z_r be the current list as defined above. The algorithm visits the point with the largest timestamp in Z_r, say p, which is a type-2 false check. Let $Z_p = U_p \cap \rho$ be the new list. We compare the new list Z_p with the old list Z_r. Notice that every point $q \notin Z_r$ with time $q.t \in [b-\tau, p.t]$ has $f(q) < f(p)$ (1), otherwise Z_r would not be in the correct top-k list. Furthermore, p is a false check because there are at least k points in $[p.t-\tau, p.t)$ with score more than the score of p, (2). From (1), (2) it follows that $Z_p \subset Z_r$. Hence, the cardinality of the new Z list is less than the cardinality of the previous Z list. In addition, notice that there are at least $k - |Z_p|$ points in $[p.t - \tau, b - \tau]$ with scores greater than the score of p, and generally greater than the score of any point in $P[b - \tau, p.t] \backslash Z_p$, (3).

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

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

Overall, with an efficient top-k module, T-Hop answers a durable top-k query $\mathsf{DurTop}(k, I, \tau)$ in $O\big((|S| + k\lceil \frac{|I|}{\tau} \rceil)(q(n) + k) \log n\big)$ time. Compared to T-Base, T-Hop runs in sublinear query time (assuming that the ratio $\lceil \frac{|I|}{\tau} \rceil$ is not arbitrarily large), i.e., the running time does not have a linear dependency on the number of records in I. Our experimental results in Section 4.6 suggests that T-Hop is one to two orders of magnitude faster than T-Base in practice.

Notice that the number of top-k queries performed by T-Hop depends on $|S|$ and

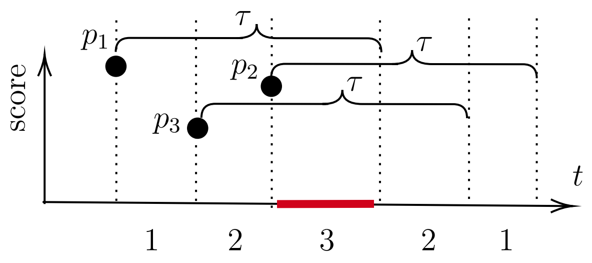


FIGURE 4.3: Blocking mechanism in score-prioritized approach

$k\lceil\frac{|I|}{\tau}\rceil$. Ideally, we would like to argue that the number of top-k queries is $O(|S|)$. In theory, the term $k\lceil\frac{|I|}{\tau}\rceil$ can be arbitrarily large comparing to $|S|$. In Section 4.5.1 we study the expected size of S in a random permutation model where a set of n scores, chosen by an adversary, are assigned randomly to the records. In such a case we show that the expected size of S is roughly $O(k\lceil\frac{|I|}{\tau}\rceil)$, meaning that in practice we expect that the number of top-k queries we execute are asymptotically equal to $|S|$.

4.4 Score-Prioritized Approach

One weakness of time-prioritized approach is that it does not pay much attention to scores and simply visit records sequentially along the timeline (with hops). Though Lemma 9 shows that T-Hop visits $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$ records in the worst case, it still potentially visits many low-score and non-durable records and ask more top-k queries. In contrast, the score-prioritized approach visits candidate records in descending order of their scores because records with high scores have a higher chance of being durable top-k records. Furthermore, these high-score records can also serve as a benchmark for future records, enabling a "blocking mechanism" to prune candidates.

Before describing the algorithms, we illustrate the main idea using an example shown in Figure 4.3. Suppose we answer a durable top-3 query with τ by visiting records in descending order of their scores: p_1, p_2 and p_3, and all three records are durable ones. p_1 has the highest score in the entire query interval, any record that lies in the τ-length time interval $[p_1.t, p_1.t + \tau]$ will be dominated by p_1, which we refer to as being "blocked" by p_1. Similarly, p_2 (the second highest score) and p_3 (the third highest score) also block a τ-length interval starting from their arrival times. The time axis is partitioned into intervals by endpoints of all blocking intervals. In Figure 4.3, the number under each interval shows how many records block this interval. Notice the bold red interval, where any record in this interval lies in three blocking intervals after processing p_1, p_2 and p_3. Since there are already three records with higher score than any record in this interval, it can not have any τ-durable top-3 record, and we can safely remove this time interval from consideration. As we continue adding blocking intervals, eventually every remaining record in the query interval will be blocked by at least three blocking intervals. The algorithm can now stop because no more durable top records can be found. The procedure is straightforwardly applicable to look-ahead version of durability, by simply reversing the direction of blocking intervals.

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

4.4.1 Score-Baseline Algorithm

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

Since all blocking intervals have the same length τ, we only need to maintain the left endpoints of such intervals (using a balanced binary search tree) to find

Algorithm 3: S-Band (k, I, τ)

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

intersection counts. The number of blocking intervals is $O(n)$. Hence, insertion and query can both be finished in $O(\log n)$ time. The sorting takes $O(n \log n)$ time so the overall query time complexity of S-Base is $O(n \log n)$.

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

*4.4.2 Score-Band Algorithm (Monotone **f** Only)*

If we could quickly find a small set of candidate records $\mathcal{C}$, which is guaranteed to be a superset of the answers; i.e., $S \subseteq \mathcal{C}$, then we could get a faster algorithm by only sorting $\mathcal{C}$. It is well-known that the k records with the highest score, with respect to any monotone scoring functions, belong to the k-skyband.[3] Hence, if a record p is τ-durable for a top-k query (with respect to a monotone f), then p must also be τ-durable for the k-skyband; i.e., p is in the k-skyband for the time interval

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

$[p.t - \tau, p.t]$. This observation enables us to construct an offline index about each record's duration of belonging to the k-skyband, and efficiently produce a superset $\mathcal{C}$ of answers to durable top-k queries. Note that the score-band algorithm has its limitation, since the k-skyband technique only applies to monotone scoring functions.

Index. Score-Band algorithm needs additional index for finding candidate set $\mathcal{C}$, which we refer to as "durable k-skyband". Suppose the value of k is known. For each record p, we compute the longest duration τ_p that p belongs to the k-skyband. Then we map each record p into the "arrival time - duration" plane as a two-dimensional point, $\tilde{p} = (p.t, \tau_p)$. We then index all such points in the 2D plane using a priority search tree [34] (or kd-tree, R-tree in practice). To answer $\mathsf{DurTop}(k, I, \tau)$, we first ask a range query with the 3-sided rectangle $I \times [\tau, +\infty]$. The set of points that fall into the search region is the superset to actual answers of durable records. This index can be constructed in $O(n \log n)$ time, has $O(n)$ space and the query time is $O(|\mathcal{C}| + \log n)$ in order to get the set $\mathcal{C}$.

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

Query Algorithm. We refer to this score-prioritized approach using durable k-skyband candidates as Score-Band algorithm, or S-Band. Full algorithm is sketched in Algorithm 3 and described below. Given k, I, τ, we first retrieve the candidate

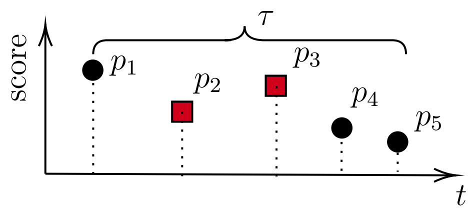


FIGURE 4.4: Durability checks in S-Band and S-Hop.

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

We can see that S-Band works similarly to S-Base. The only difference is that for a record that is blocked less than k times, we still have to execute a top-k query to check whether the record is τ-durable (Line 6). This step of durability check is necessary. Though some records are guaranteed to be non-durable (i.e., not captured by $\mathcal{C}$ with durable k-skyband), they can still block other records (with lower scores) to be durable ones. Consider a concrete example in Figure 4.4 where black dots represent candidate records in $\mathcal{C}$ and red squares represent records that are not in $\mathcal{C}$. S-Band would only visit p_1, p_4 and p_5. At the time we visit p_4, there is only one blocking interval (introduced by p_1). However, p_2 and p_3 actually have higher scores than p_4. By running a durability check query on p_4, we can discover these missing records and add corresponding blocking intervals (Line 10-11) for better pruning power in future steps.

Complexity. The query time complexity of S-Band can be decomposed into three parts: 1) a range search query to find candidate set $\mathcal{C}$; 2) sort $\mathcal{C}$ according to their scores; 3) find durable records from sorted $\mathcal{C}$ sequentially. Summing up the above, the overall query time complexity of S-Band is $O\big(|\mathcal{C}|(q(n) + k)\log n\big)$, assuming that a top-$k$ query can be answered in $O(q(n) + k)$ time. In the worst case $|\mathcal{C}| = O(n)$ since all points can lie in the k-skyband. In Section 4.5 we show that using the probabilistic model in [13] (where the coordinates of the points are randomly assigned) the expected size of $\mathcal{C}$ is $O(k\lceil \frac{|I|}{\tau} \rceil \log^{d-1} \tau)$. Due to the blocking mechanism, in practice we expect that the number of top-k queries will be smaller. However, notice that we always need to sort all records in $\mathcal{C}$ which might make S-Band much slower due to the size of $\mathcal{C}$ that increases (in expectation) exponentially on the dimension d.

4.4.3 Score-Hop Algorithm

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

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

the Time-Hop algorithm, since we effectively skip certain records while we traverse records in descending order of their scores, as we taking a hop in the score-domain.

Query Algorithm. We refer to this solution as Score-Hop algorithm, or S-Hop. The full description of the algorithm is shown as follows, and also in Algorithm 4.

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

Compared to S-Band, S-Hop does not have a strong dependency on the dimension of the data (only the running time of the top-k queries depends on the dimension) and makes better use of the blocking mechanism. In the end, we only find and process high-score records as we need instead of acquiring a full sorted order of records in advance, which leads to better worst case theoretical guarantees and faster query time. Experimental results in Section 4.6 demonstrate that S-Hop can be 1 to 2 orders of magnitude faster than S-Band on high-dimensional ($\geqslant 10$) datasets.

Correctness. The following lemma proves the correctness of S-Hop; full proof can be found in Appendix A.1.

Lemma 10. *Given k, I and τ, the Score-Hop algorithm returns the correct answer for durable top-k query.*

Proof (Sketch). Let S^* be the τ-durable records in I. We show that $S \subseteq S^*$ and $S^* \subseteq S$. The algorithm always checks by running a top-k query if a record should be in the solution (line 8 of Algorithm 4) so $S \subseteq S^*$.

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

4.4.4 Complexity Analysis of S-Hop

The query complexity analysis of S-Hop is non-trivial and needs more care. There are three main sub-procedures in S-Hop: find next highest score record, top-k queries for durability check and blocking mechanism. As presented above, the first two components both rely on multiple top-k queries. We first show a worst-case guarantee on the total number of top-k queries called in the algorithm. For simplicity, we only sketch the proof. See Appendix A.1 for full proofs.

Lemma 11. *The total number of top-k queries performed by the Score-Hop algorithm is $O(|S| + k\lceil\frac{|I|}{\tau}\rceil)$.*

Proof (Sketch). As we had in the proof of Lemma 9 we need to bound the number of false checks. Let p be a false check and let p' be the record with the largest timestamp in $Q(k, [p.t - \tau, p.t])$. We say that p is assigned to p'. If $p'.t < a$, where a is the timestamp such that $I = [a, b]$, then we assign p to a. We first show that at the moment that we find the false check p the corresponding record p' can only have one of the following three properties: i) it lies in at least k blocking intervals,

ii) $p' \in S$ and it lies in at most $k-1$ blocking intervals, iii) $p' = a$. If p' has property ii) then p is a type-1 false check. Otherwise, p is a type-2 false check.

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

The lemma above also shows that the number of different sets M_j that are created by the algorithm is $O(|S|+k\lceil \frac{|I|}{\tau} \rceil)$. For each set we can visit at most k records so in total the algorithm may visit $O(k(|S| + k\lceil \frac{|I|}{\tau} \rceil))$ records [4]. Each top(), or pop() procedure takes $O(\log n)$ time so in total we need $O(k(|S| + k\lceil \frac{|I|}{\tau} \rceil) \log n)$ to visit these records. Furthermore, recall that we need $O(\log n)$ time to check if a record lies in at least k

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

blocking intervals (using a binary search tree) so we also spend $O(k(|S|+k\lceil\frac{|I|}{\tau}\rceil)\log n)$ time for the blocking mechanism. Notice that this running time is dominated by the time to answer $O(|S|+k\lceil\frac{|I|}{\tau}\rceil)$ top-k queries, so S-Hop answers a durable top-k query in $O\big((|S|+k\lceil\frac{|I|}{\tau}\rceil)(q(n)+k)\log n\big)$ time (with an efficient top-k query procedure in $O(q(n)+k)$).

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

4.5 Expected Complexity

In the previous sections we presented two types of algorithms (time-prioritized and score-prioritized) to answer durable top-k queries with the same worst-case guarantees on their query time. In particular we showed that their query times depend on $k\lceil\frac{|I|}{\tau}\rceil$ and $|S|$. In this section, we go beyond the worst-case analysis and analyze their performance in a more "expected" sense. Most importantly, we show in Section 4.5.1 that the expected size of $|S|$ is roughly $k\lceil\frac{|I|}{\tau}\rceil$ if the scores of data records are drawn randomly from an arbitrary distribution (which can be picked by a powerful adversary with the advance knowledge of the query parameters). This result essentially establishes that, under this model, our best algorithms are in a sense optimal because their complexity is expected to be linear in the output size. Secondly, in Section 4.5.2, to study the expected complexity of Score-Band algorithm in Section 4.4.2 by bounding the expected size of τ-durable k-skyband candidate set $\mathcal{C}$ using the same probabilistic model used in [13] to bound the expected number of skyline points.

4.5.1 Expected Answer Size

Consider a set of n records P with $p_i.t = i$, for $p_i \in P$. We analyze the expected size of a query output when the score of records are assigned in a semi-random manner, where the data values can be arbitrarily chosen and then they are assigned in a random order to the records. More formally, we consider a random permutation model (RPM). Let $\mathbf{X} = x_1 < x_2 < \ldots < x_n$ be a sequence of n arbitrary non-negative numbers chosen by an adversary, and let σ be a permutation of $\{1, \ldots, n\}$. We set $f(p_i) = x_{\sigma(i)}$, i.e., the score of record p_i is $x_{\sigma(i)}$, where $\sigma(i)$ is the image of i under σ. As argued in [8], the random permutation model is more general than the model in which all scores are drawn from an arbitrary unknown distribution, so our result holds for this model as well. The random permutation model has been widely used in a rich variety of domains and considered as a standard for complexity analysis; i.e., online algorithms [48, 77, 81], discrete geometry [7, 6, 52], and query processing [8]. Our main result is the following.

Lemma 12. *In the random permutation model, given k, τ and I, we have $\mathbb{E}[|S|] = k\frac{|I|}{\tau+1}$.*

Proof. For a record $p_i \in P(I)$, let X_i be the random variable, which is 1 if p_i is a τ-durable record, and 0 otherwise. Thus, $\mathbb{E}[|S|] = \mathbb{E}[\sum_i X_i]$. Using the linearity of expectation, $\mathbb{E}[\sum_i X_i] = \sum_i \mathbb{E}[X_i] = \sum_i \mathbf{Pr}[X_i = 1]$.

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

$$\mathbf{Pr}[X_i = 1] = \sum_{l=0}^{k-1} \sum_{Q \subset P_i^\tau, |Q|=l} \mathbf{Pr}[A_Q]. \tag{4.1}$$

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

$$\mathbf{Pr}[A_Q \mid V] = \frac{l!(\tau - l)!}{(\tau + 1)!} = \frac{1}{\tau + 1} \frac{1}{\binom{\tau}{l}}. \tag{4.2}$$

Since (4.2) holds for all V, $\mathbf{Pr}[A_Q] = \frac{1}{\tau + 1} \frac{1}{\binom{\tau}{l}}$. Substituting this in (4.1), we obtain

$$\mathbf{Pr}[X_i = 1] = \sum_{l=0}^{k-1} \binom{\tau}{l} \frac{1}{\tau + 1} \frac{1}{\binom{\tau}{l}} = \sum_{l=0}^{k-1} \frac{1}{\tau + 1} = \frac{k}{\tau + 1} \tag{4.3}$$

Finally,

$$\mathbb{E}[|S|] = \sum_i \mathbf{Pr}[X_i = 1] = k \frac{|I|}{\tau + 1}. \tag{4.4}$$

□

Combining Lemma 12 with the analysis of Sections 4.3.3 and 4.4.4, we conclude that in a random permutation model the expected query time complexity of both Time-Hop and Score-Hop algorithms is $O(|S|(q(n) + k)\log n)$, or equivalently $O\big(k\lceil\frac{|I|}{\tau}\rceil(q(n)+k)\log n\big)$, where $O(q(n)+k)$ reflects the time complexity of answering a top-k query. In Section 4.6, our experimental results on real and synthetic datasets both confirm this finding.

Table 4.1: Dataset summary

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

4.5.2 Expected size of durable k-skyband

In this subsection we bound the expected size of τ-durable k-skyband records, denoted by $\mathcal{C}$, from Section 4.4.2 in a probabilistic model similar to the previous case. Recall that the size of $\mathcal{C}$ affects the running time of the S-Band algorithm.

Let $P = \{p_1, \ldots, p_n\}$ with $p_i.t = i$. We use the same random model as in [13] where (the attributes of) records are randomly generated. The following lemma bounds the expected size of $\mathcal{C}$.

Lemma 13. *In the random model as in [13], given k, τ and I, we have $\mathbb{E}[|\mathcal{C}|] = O(k\frac{|I|}{\tau}\log^{d-1}\tau)$.*

Combining Lemma 13, the analysis of Section 4.4.2 and an efficient top-k query procedure runs in $O(q(n) + k)$ time, the expected query time complexity of Score-Band algorithm is $O\big(k\lceil\frac{|I|}{\tau}\rceil(q(n) + k)\log n \log^{d-1}\tau\big)$. It shows that the expected complexity of Score-Band algorithm can be higher than Time-Hop or Score-Hop algorithm by a factor of at most $\log^{d-1}\tau$. Experimental results in Section 4.6 also confirm this finding as we vary the data dimensionalities. The curse of dimensionality makes Score-Band algorithm perform worse even compared to other simple baselines. Again, Time-Hop and Score-Hop are both generally applicable to arbitrary user-specified scoring functions, while Score-Band only works for monotone functions.

4.6 Experiments

4.6.1 Experiment Setup

Datasets. We use two real-life datasets and some synthetic ones, as summarized in Table 4.1 and described below:

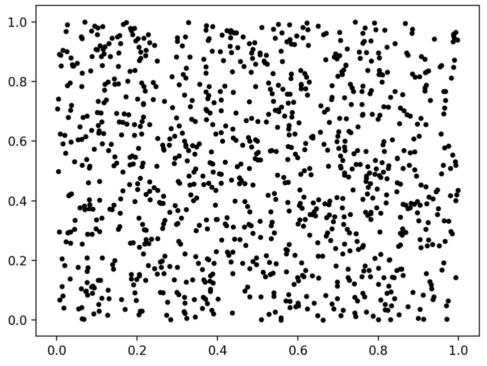
NBA[5] contains the performance of each NBA player in each game from 1983 to 2019, with in total $\sim$ 1 million individual performance records on 15 numeric attributes. Records are naturally organized by date and time, and we break ties (e.g., performances of different players in the same game) arbitrarily.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

Network[6] is the dataset from KDD Cup 1999. This dataset contains $\sim$ 5 million records with 37 numeric attributes that describe network connections to a machine, including connection duration, packet size, etc. Records have unique timestamps and are ordered by these timestamps. Since these attributes have different measurement units, we scale the value of each dimension using Min-Max normalization. To study the impact of data dimensionalities on query efficiency, we choose the first 2, 3, 5, 10, 20, 30 and 37 attributes from the full dimensions to create 7 different datasets collectively referred to as "Network-X", where **X** represents the dimensionality of the dataset.

Syn is a synthetic two-dimensional dataset that is used for scalability test on proposed solutions. We generate Syn with independent (IND) and anti-correlated (ANTI) data distributed in a 2D unit square. For IND data, the attribute values

[5] NBA datasets were collected from `https://www.basketball-reference.com/`

[6] `https://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`


(a) IND

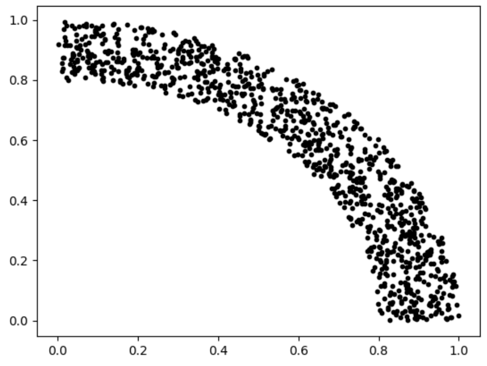

(b) ANTI

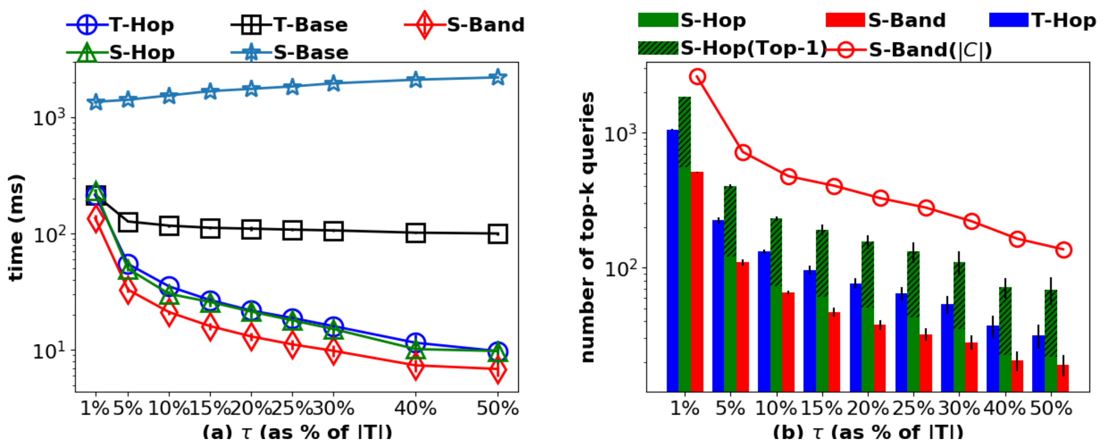
FIGURE 4.5: Value distributions for synthetic dataset

Table 4.2: Query Parameters (default value in bold)

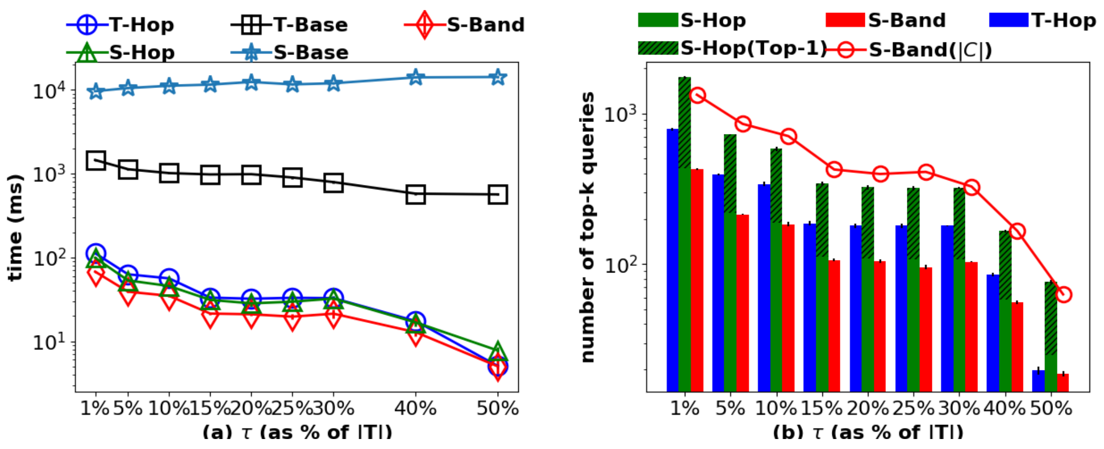
Parameter	Range
k	5, **10**, 15, 20, 25, 30, 35, 40, 45, 50
τ	1%, 5%, 10%, 15%, **20%**, 25%, 30%, 40%, 50%
$\|I\|$	10%, 20%, 30%, 40%, **50%**, 60%, 70% 80%
d	1, **2**, 3, 5, 10, 20, 30, 37

of each tuple are generated independently, following a uniform distribution. ANTI data are drawn from the portion inside the positive orthant of an annulus centered at the origin with outer radius 1 and inner radius 0.8, representing an environment that most of the records gather in k-skyband. Figure 4.5 illustrates the sample value distributions of IND and ANTI. The full size of Syn is 50 million and each data point has an unique arriving time. We further choose several subsets of Syn with 1, 2, 5, 10 and 20 millions of records. The set of synthetic datasets are collectively referred to as "Syn-X", where **X** represents data size.

Query Parameters. Table 4.2 summaries the query parameters under investigation, along with their ranges and default values. Among these, the query interval length $|I|$ and the durability τ is measured as percentage of dataset size n. When varying query interval length, we always fix the right endpoint of the interval to be



(a) Performance on NBA-2 as τ varies.

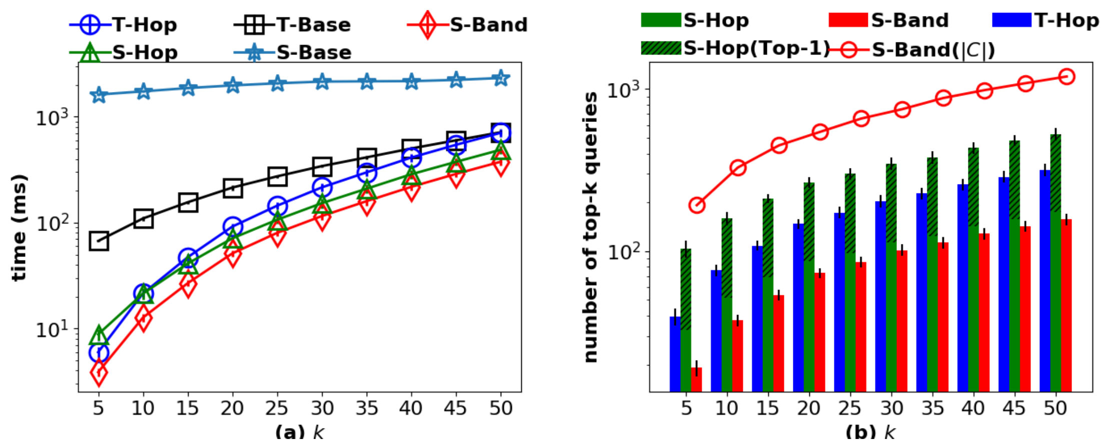


(b) Performance on Network-2 as τ varies.

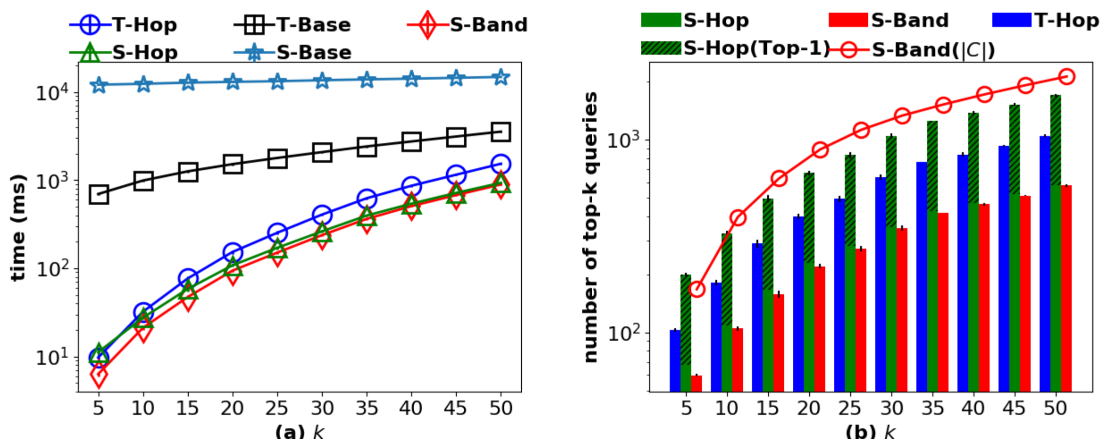
FIGURE 4.6: Performance comparison as τ varies.

the most recent timestamp in dataset and only move the left endpoint.

Implementations & Evaluation Metric. To make the discussions concrete and concise, we choose a linear and monotone preference scoring function throughout the experimental section in the simple form: $f(p) = \sum_{i=1}^{d} \mathbf{u}_i \cdot p.x_i$, where $\mathbf{u}$ is a user-specified preference vector and $\mathbf{u}_i$ is the (non-negative) weight for i-th attribute of a record. At query time, user need to specify $\mathbf{u}$ as one of the input parameters. Since the focus of this chapter is not to develop the best possible index for top-k queries $Q_{\mathbf{u}}(k, W)$, our implementation of the top-k building block simply adopts a tree index (on the time domain of P), and answer $Q_{\mathbf{u}}(k, W)$ in a straightforward top-down



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



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

FIGURE 4.7: Performance comparison as k varies.

manner with a branch-and-bound method. This index offers adequate performance in our experiments, but it can certainly be replaced by more sophisticated index with better worst-case guarantees, without affecting the rest of our proposed solution.

Using the building block of top-k queries described above, we further implement T-Base (Section 4.3.1), T-Hop (Section 4.3.2), S-Base (Section 4.4.1), S-Band (Section 4.4.2) and S-Hop (Section 4.4.3). Performance of various methods are evaluated using the following two metrics: number of top-k queries and overall query time (in millisecond). For each query parameter setting, we run the query 100 times with 100 different randomly generated preference vectors, and report the average with standard deviation.

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

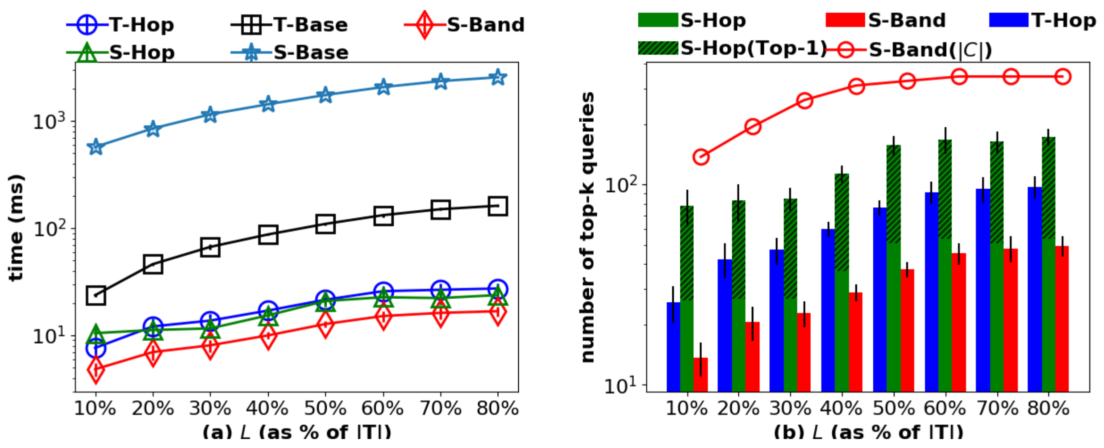
4.6.2 Algorithm Evaluations

According to the theoretical analysis of our algorithms in previous sections, the query efficiency depends on the length of durability window τ, the value of k, the length of query interval I, the data dimensionality d and the data size n. For fair evaluation and comparison of algorithm efficiency, we designed a set of variable-controlling experiments such that each time we only vary one query parameter of interest and fix the others to default values.

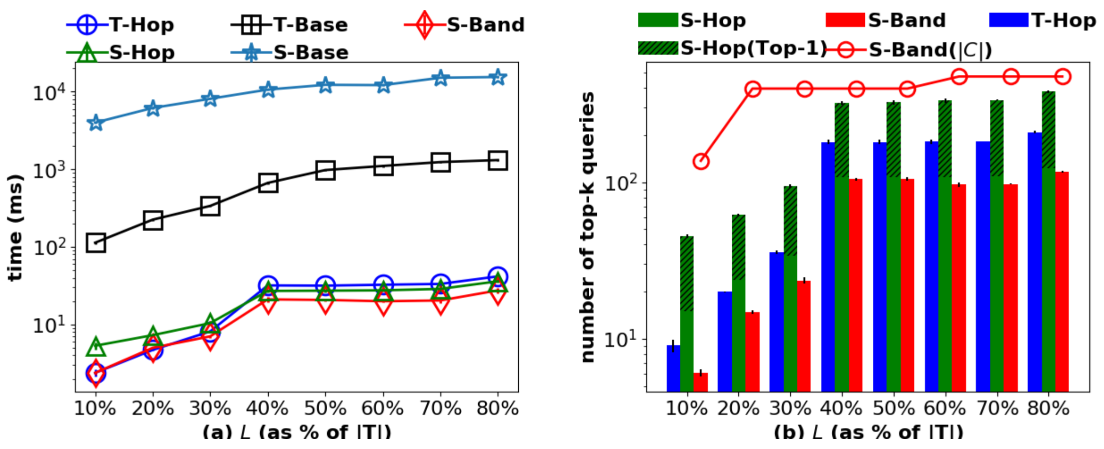
Comparison of Algorithms when Varying τ. In Figure 4.6, we investigate the performance of all durable top-k solutions, as we vary durability τ. Figure 4.6-1-(a) shows the query efficiency comparison on NBA-2. The sorting based solution S-Base is the slowest, as it requires fully sorting all records in the time interval of length $|I| + \tau$. T-Base is faster than S-Base and mostly independent of τ. All the rest solutions, T-Hop, S-Hop and S-Band, become more efficient as we increase τ, or equivalently, when query is more selective. This finding confirms our analysis in Section 4.5 that the query efficiency bounds of Hop-based solutions and S-Band both depend on the answer size, which is $O(k\frac{|I|}{\tau})$. T-Hop and S-Hop nearly perform the same, while S-Band can be slightly faster. When the query is highly selective (τ is half of the length of entire time domain), they are 1-2 orders of magnitude faster compared to T-Base and S-Base, respectively. Similar trends can be seen in Figure 4.6-2-(a), where we test algorithms on a larger dataset Network-2. The only difference is that baseline solutions (T-Base and S-Base) are more expensive and the efficiency difference between baseline solutions and T-Hop/S-Hop/S-Band is even larger (up to 3 orders of magnitude).

Next, we take a closer look at T-Hop, S-Hop and S-Band in Figure 4.6-1-(b), which compares the number of top-k queries needed for these three advanced algorithms. For S-Hop, the total number of top-k queries is decomposed into two parts: top-k queries for durability check (unshaded region of a green bar) and top-k queries for finding the next highest score record (shaded region). For S-Band, we also plot the size of durable k-skyband candidate set C on top the figure as red circled line, reflecting the overhead cost of sorting C for S-Band. Now it is clear that the main reason why T-Hop/S-Hop/S-Band becomes faster when τ is large is that fewer top-k queries are needed. A more selective query with larger τ also makes the candidate set C of S-Band smaller, demonstrating the effectiveness of using durable k-skyband to identify promising candidates. On the other hand, we can see that S-Hop and S-Band ask fewer top-k queries than T-Hop, demonstrating the pruning power of blocking mechanism in score-prioritized solutions. This figure also explains why S-Band runs slightly faster than S-Hop and T-Hop on NBA-2 in this case, as S-Band requires the least number of top-k queries and the overhead cost on sorting candidate set C is relatively small on two-dimensional data. Again, similar trends can be found in Figure 4.6-2-(b).

Comparison of Algorithms when Varying k. Next, we study the effect of k on efficiency. Results are shown in Figure 4.7. When we increase k, not only need we ask more top-k queries (see Figure 4.7-1-(b) and Figure 4.7-2-(b)), but a top-k query itself also becomes more expensive. Thus in both Figure 4.7-1-(a) and Figure 4.7-2-(a), we can see that all algorithms (except S-Base) are slower when k is larger. Especially when k reaches 50, top-k computations become the dominant factor on overall efficiency, and the differences among the various algorithms diminish. Still, S-Band and S-Hop have slight advantages over T-Hop on larger k, as they use blocking



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

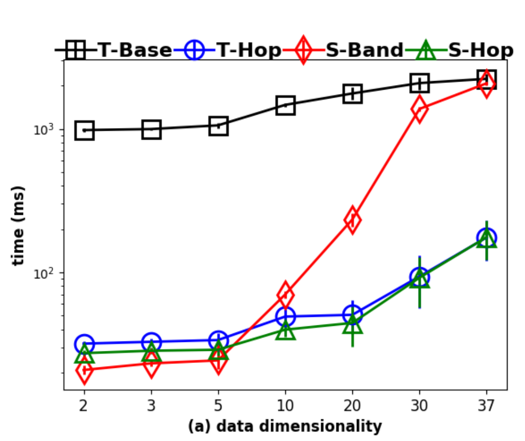


(b) Performance on Network-2 as $|I|$ varies.

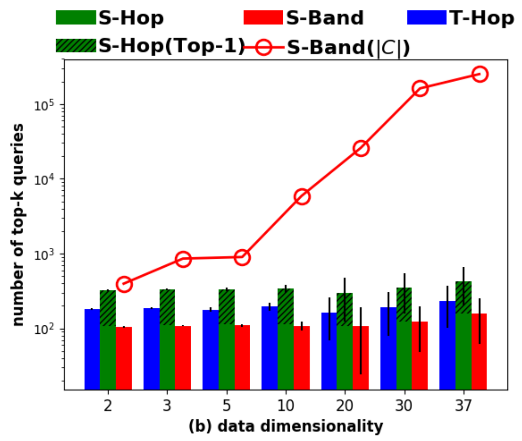
FIGURE 4.8: Performance comparison as $|I|$ varies.

mechanism to prune candidate records and are more conservative in asking expensive top-k queries.

Comparison of Algorithms when Varying $|I|$. In Figure 4.8, we compare the performance of proposed algorithms as we vary the query interval length $|I|$. In terms of efficiency, Figure 4.8-1-(a) and Figure 4.8-2-(a) show that T-Hop/S-Hop/S-Band is much faster than baseline solutions T-Base and S-Base, especially on the large dataset Network-2. On the other hand, we also find that our proposed algorithms scale better with $|I|$ than with k (recall Figure 4.7). The reason is that the time complexities of T-Hop/S-Hop and S-Band are quadratic in k but only linear on $|I|$



(a)

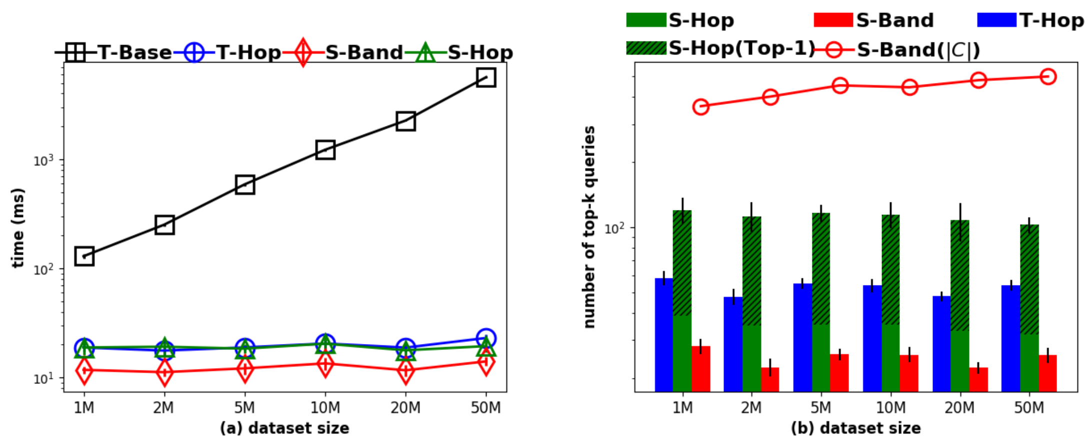


(b)

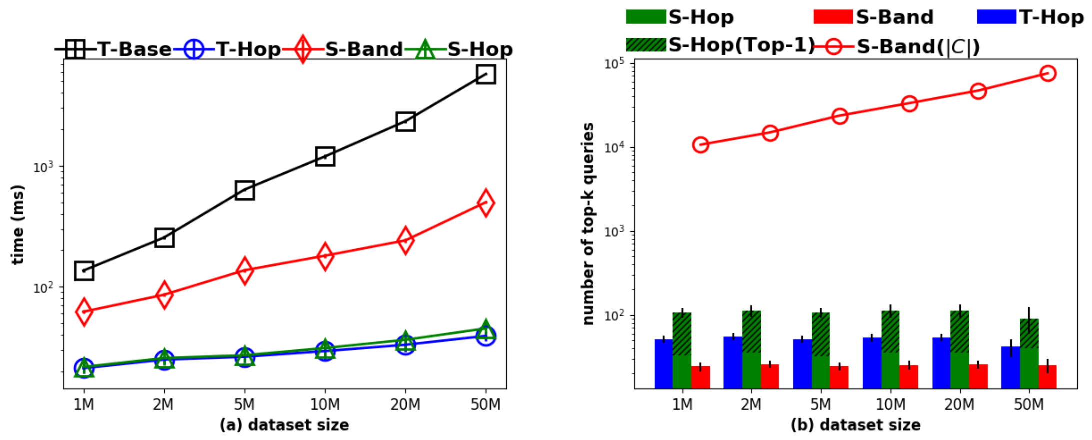
FIGURE 4.9: Performance comparison on Network-X as d varies.

(recall Lemma 12 and Lemma 13). In terms of number of top-k queries, in Figure 4.8-1-(b) and Figure 4.8-2-(b), it is not surprising to see that all proposed solutions ask more top-k queries as $|I|$ increases. The relative performance of various algorithms is consistent with previous experiments where we varied τ or k.

Comparison of Algorithms when Varying d. In this section, we study the effect of data dimensionality d on algorithm performances. Since the sorting-based S-Base is clearly inferior to other algorithms, here we only test T-Base, T-Hop, S-Band and S-Hop on Network-X with varying dimensions. Results are shown in Figure 4.9. Let us first take a look on Figure 4.9-2. We can see that the number of top-k queries for all proposed algorithms stays stable as we increase dimensionality. This finding again confirms our theoretical analysis that the number of top-k queries (or, answer size) depends only on $k\frac{|I|}{\tau}$ and is independent of dimensionality d. On the other hand, we can see that the size of candidate set C for S-Band rockets in high dimensions, and can be up to 4 orders of magnitude larger than the size of actual promising records. The sorting overhead on such huge candidate sets is already too big. Then, let us go back to Figure 4.9-1. The query time of T-Base, T-Hop and S-Hop slowly increases as we increase dimensionality, because top-k queries on high-dimensions



(a) IND



(b) ANTI

FIGURE 4.10: Scalability test on IND and ANTI Syn-X.

become more expensive, yet they ask roughly the same number of top-k queries regardless of dimensionality. While S-Band still performs well on low-dimensional data (less than 5 dimensions), in higher dimension S-Band becomes dramatically worse, even taking as much time as T-Base on Network-37.

Scalability. Finally, we use the two-dimensional synthetic dataset Syn-X to test the scalability of the proposed algorithms as we vary the input size from 1 million to 50 million. Figure 4.10 summarizes the results. As the input size increases, we also increase the query interval length proportionally (so it remains at a fixed percentage of the data size). As shown in Figure 4.10-1, we can see that T-Hop, S-Hop and S-Band scale well on large IND datasets, and S-Band again performs slightly better than T-Hop and S-Hop. The running time of S-Base increases on larger datasets simply because we are also making the query interval longer. Figure 4.10-1-(b) further illustrates that the total number of top-k queries asked by different algorithms is also independent from the data size. A larger dataset only makes top-k queries more expensive. Although the size of candidate set $|C|$ increases on larger IND datasets, its growth rate here is much lower than its growth rate when varying dimensionality d in Figure 4.9. Overall, on IND synthetic data, $|C|$ is only about 4-5 times bigger than the actual answer size, which will not incur a big sorting overhead for S-Band. However, the situation is much different for ANTI Syn-X. As shown in Figure 4.10-2, in terms of query efficiency, T-Hop and S-Hop still scale well, but S-Band now becomes much more expensive because of the data distribution of ANTI. Most records in ANTI data would gather in k-skyband, resulting in C up to 3 orders of magnitude larger than the actual answer size (see Figure 4.10-2-(b)), which hurts the performance of S-Band. The efficiency of S-Band has a strong dependency on the candidate set C, or more generally, the data distribution. In contrast, the performance of T-Hop and S-Hop in this case is nearly independent of both size and distribution of data; it is only linear to the answer size.

Query Time Distribution over Different Real Datasets. Figure 4.10 already clearly illustrates the performance difference of S-Band on IND and ANTI synthetic data, demonstrating the effect of data distributions on S-Band's query efficiency. Here, we further compare T-Hop, S-Hop and S-Band on real data, and study how data distributions would influence their performance in practice. We use NBA as the main data source, and select 20 combinations of 5 dimensions randomly chosen out of the 15 attributes, e.g., (points, assits, rebounds, steals, blocks), (points, assits, steals, blocks, 3-pointers-made), etc. These resulting 20 datasets have the same

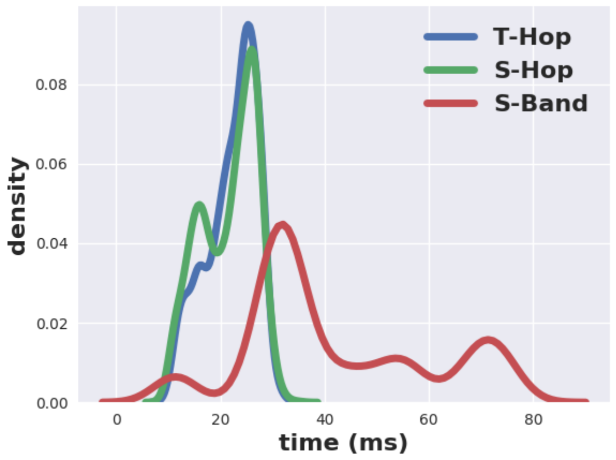


FIGURE 4.11: Runtime distribution on 5d NBA data.

Table 4.3: Query time (in seconds) comparison on NBA-2 when varying τ. PostgreSQL backend.

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

dimensionality (5) but exhibit different distributions. We run queries with default settings on each dataset, and plot the running time distribution for all datasets. Results are shown in Figure 4.11. We can see that S-Band takes longer time on average, and also has a wide span on query time. This finding again confirms that S-Band is highly sensitive to underlying data distributions. In contrast, running times of T-Hop and S-Hop are centered in narrower value ranges, showing their robustness to data distributions and further demonstrating their advantages over S-Band on real data.

4.6.3 DBMS-Based Implementations

To demonstrate the generality of proposed solutions and its possibility of integrating into a DBMS, we further test the algorithms utilizing PostgreSQL [1] as the backend DBMS. More specially, we load the datasets NBA-2, Syn-500M (IND) and Syn-500M

Table 4.4: Query time (in seconds) comparison on NBA-2 when varying L. PostgreSQL backend.

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

Table 4.5: Query time (in seconds) comparison on different datasets. Dataset size (measured by DBMS storage size) is shown in parentheses. PostgreSQL backend.

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

(ANTI) into PostgreSQL tables. The table schema consist of numeric attributes of the records and an additional column representing arriving time instant. For algorithm implementations, we code T-Hop and T-Base as stored procedures using PL/Python with PostgreSQL's native support operators.[7] Besides data tables, we also create corresponding index tables to support efficient top-k records retrieval. The index table is similar to the tree-based index as we used for previous experiments, providing sufficient data reduction for answering range top-k queries. Again, the top-k module can certainly be replaced by more sophisticated index with better performance, without affecting the rest of our proposed solution.

Tables 4.3 and 4.4 show the results of testing T-Hop and T-Base on the smaller NBA-2 dataset with the same query setting as before, varying durability τ and query interval length L to compare query efficiencies. Similar conclusions can be drawn here. T-Base always pays linear cost (continuous sliding windows) to visit all records in the query interval. Thus, the running time is linear to L (Table 4.4), and nearly independent of τ (Table 4.3). In comparison, T-Hop's complexity is linear to the answer size, which makes it run faster as query becomes more selective (smaller L

[7] The other proposed solution, S-Hop, requires a more delicate query procedure and data structures (recall Algorithm 4). Hence it is more suitable to implement S-Hop as a wrapper function outside the DBMS, instead of a stored procedure.

or larger τ). Overall, T-Hop is at least 10× faster than T-Base.

In Table 4.5, we increase the dataset size up to 500M records, which takes around 30 Gigabytes of disk space in PostgreSQL. Running default queries in such cases, we can see that T-Hop is more than 100× faster than T-Base, bringing down the query time from nearly 12 minutes to just 2 seconds. T-Hop also apparently scales well on large datasets, since the complexity is mostly linear to the answer size. The query time increase solely comes from the more expensive top-k module. On the contrary, the continuous sliding-window nature of T-Base makes it prohibitively slow when dealing with large amounts of temporal data.

4.6.4 Summary of Experiments

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

4.7 Related Work

The notion of "durability" on temporal data has been studied by previous works, but they consider different definition of durability and/or different data models from ours. In [65] and [118], authors implicitly considered "durability" in the form of prominent streaks in sequence data, and devised efficient algorithms for discovering such streaks. Given a sequence of values, a prominent streak is a long consecutive subsequence consisting of only large (small) values. An example prominent streak would be "This month the Chinese capital has experienced 10 days with a maximum temperature in around 35 degrees Celsius – the most for the month of July in a decade." One important observation is that prominent streaks are skyline in two dimensions– streak interval length and minimum value in the interval. Their solutions hence focused on generating limited promising streak candidates and then performing skyline operator on candidates. Their algorithms can also be extended to find general top-k, multi-sequence and multi-dimensional prominent streaks. Jiang and Pei [64] studied Interval Skyline Queries on time series, which can be viewed as another type of "durability" when segments of time series dominate others.

Another line of durability-related work on temporal data are represented by [73, 108, 41] and [78]. Consider a time-series dataset with a set of objects, where the data values of each object are measured at equal time interval; i.e., stock markets. At each time instance t, Objects are ranked according to their values at t. Authors in these work considered the definition of "durability" as the fraction of time during a given time window when an object ranks k or above. For example, a durable query here would ask for "stocks that were among the top 5 most heavily traded for at least 80% of the trading days during the last quarter of 2017". This line of work mainly focused on how to efficiently aggregate rankings (rank $\leqslant k$ or not) over time. In [73], the authors proposed a compacted format using bit maps to store rankings per object at each timestamp. [108] proposed to only remember rank changes between consecutive timestamps of each object under the assumption that ranks of temporal objects tend to stay stable over time. Gao et al. [41] considered use prefix sum to accumulate the total times that an object ranks k or above (given a specific value of k) for constant time durability computation. To cap storage, authors proposed to select only a small portion of prefix sums such that it can maximize the accuracy of approximate answers according to given query distributions. [78] applied durable top-k searches in document archives, finding the set of documents that are consistently among the most relevant to a set of keywords throughout a given time interval. Under this setting, the difficulty is how to merge multiple per-keyword relevance scores over time efficiently into a single rank among others.

Durable queries also arise in dynamic graphs or temporal graphs, where graphs evolve over time and are typically represented as a sequence of graph snapshots. For example, in [96] and [97], authors considered the problem of finding the (top-k) most durable matches of an input graph pattern query; that is, the matches that exist for the longest period of time. The main focus is more on the representations and indexes of the sequence of graph shots, and how to adapt classic graph algorithms in a time-varying setting.

Interestingly, persistent homology in computational topology [37] similarly represents the notion of durability as in temporal data. More persistent features (detected over a wide range of spatial scales) are more likely to represent true features of the underlying space rather than artifacts of sampling, noise, or particular choice of parameters.[8]

Besides durability, Mouratidis et al. [82] studied the problem of continuously monitoring top-k results over the most recent data in a streaming setting. Our

[8] `https://en.wikipedia.org/wiki/Persistent_homology`

baseline solution used in Section 4.6 shares the same spirit as algorithms in [82] for incrementally maintaining top-k results over consecutive sliding windows.

4.8 Conclusion

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

5

Durability Queries on Probabilistic Temporal Data

The longer you can look back, the farther you can look forward.

Winston Churchill

5.1 Introduction

Chapter 3 and Chapter 4 discuss durability queries with ranking operations on sequence-based temporal data and instant-stamped temporal data, respectively. One commonality between these two works is that they both analyze existing historical temporal data. Going beyond, as Example 3 shows, the notion of durability can be also extended into the future, where we do not have the temporal data exactly, but only in probabilistic representations.

In this chapter, we study the problem of answering another type of durability queries, which is in the form of statistical prediction queries, on probabilistic temporal data. Intuitively, the notion of durability predicts how long (or, how likely) a condition that currently holds will remain valid in the future.

When predicting the future, people naturally ask such kind of queries, generalizing from Example 3. For instance, business analysts might ask durability queries for risk assessments: "what is the probability that our financial product will keep a 30% profit margin in the next quarter?" or "what is the probability that our client will not have a default on his mortgage loan in the next five years?" In some critical design and testing scenarios, e.g., self-driving cars, we can also see durability queries: "what is the probability that a Tesla in full auto-pilot mode has no misjudgements within 500 miles during rainy days?" As can be seen from aforementioned examples, there are two technicalities of durability prediction queries. First, our query predicts the future, where the underlying data can only be probabilistic or inferred from existing historical data. Query processing over probabilistic data poses new challenges compared to that on deterministic data. Second, temporal dependence broadly exists in temporal data, especially in domains like financial market or user behaviors where the present is believed to have strong dependence on the recent past. This distinguishes our work from previous studies on query processing over probabilistic databases [89, 31, 101, 60, 115, 59, 46], where data uncertainty is only considered independently for each individual object (e.g., attribute-level or tuple-level). To deal with data uncertainty and temporal dependence, we assume there exists a stochastic process that probabilistically represents the future, where it provides step-by-step forward predictions of a temporal series into the future. Such models are commonly used in practice for temporal data modelling and prediction, ranging from classic statistical models like Auto-Regressive model (AR) [99] and Markov Chains [69], to more complex deep learning models such as the Recurrent Neural Network (RNN) [93, 67] and its variants [55].

Given any stochastic process that allows generation of future data, [1] we can an-

[1] How to choose and derive a suitable and accurate stochastic process from historical data is an interesting and important problem, but beyond the scope of this paper.

swer durability queries by running Monte Carlo simulations [14] and estimating the probability that a certain event happens. However, it is well known that the standard Monte Carlo (MC) technique, i.e., Simple Random Sampling, suffers from its inefficiency to produce reliable estimate, especially when dealing with small probabilities. Considering the practical use cases of durability queries, where people are more interested in looking for robust and consistent behaviors in the future, the query answers are mostly small probabilities (that breaks the current condition). Such nature of durability queries further amplifies the drawbacks of the standard MC. As pointed out in multiple literature [72, 85], the relative error of standard MC increases to infinity as the underlying probability approaches 0, resulting in prohibitively expensive cost for these scenarios.

To meet the challenges of durability queries on probabilistic temporal data, this chapter proposes an alternative sampling approach that shares the same merits as the standard MC, such as simplicity and generality, but provides significant efficiency improvement. The basic idea of our approach is similar to importance sampling [92] that directs simulation efforts more towards the target. Given a stochastic process that exhibits some continuity and dependency on the past, an important observation is that processes whose states are "closer" to the target have relatively higher probabilities to finally reach the target. Motivated by this observation, our approach would set several intermediate goals between the starting point and the target, and continuously reward the simulations that reach these milestones by running additional simulations that continue from these states. By doing this, our approach effectively directs the simulations towards the desired sampling distribution and allocates more simulation effort to those "promising" simulation traces. Note that we could achieve this goal simply by running simulations based on the underlying temporal model, which generally views the model as a black-box and minimizes our solution's dependence on the model to the greatest extent possible.

To summarize, we have made the following contributions:

- We propose to apply an alternative sampling approach called Multi-Level Splitting Sampling (MLSS) for durability query processing. The new solution combines the idea from importance sampling and branching process theory. We prove how the new sampler leads to an unbiased estimator and analytically derive its variance complexity.

- We go beyond the standard MLSS, which has been well studied but has certain restrictions, and propose a novel and general MLSS procedure, making it widely applicable to any stochastic process. We also prove the unbiasedness of general MLSS and analyze its variance.

- We further present an adaptive greedy strategy that automatically search for (near-) optimal parameters of MLSS with small search overhead, simply through simulations. The proposed approach frees users from time-consuming parameter tuning process when using MLSS in practice.

- Extensive experiments on various real-life applications, involving both classic statistical models and complex deep learning models, demonstrate the efficiency of our solutions over existing standard techniques. Our best solution provides up to an order-of-magnitude query time speedup, without sacrificing of answer quality.

To the best of our knowledge, this work is among the first to study durability queries (or more generally, statistical prediction queries) from generative temporal models. Though this chapter mainly focuses on durability query processing, our proposed techniques are general and can be straightforwardly applied to efficiently answer broader types of statistical queries (with small probabilities) that involve Monte Carlo simulations.

The rest of the chapter is organized as follows. Section 5.2 formally defines durability queries, and reviews several key concepts and existing solutions to the problem. Section 5.3 elaborates the standard MLSS, including a sampler and estimator together with its complexity analysis. Section 5.4 further introduces a novel and general MLSS procedure. Section 5.5 discusses MLSS's empirical optimizations and presents an adaptive greedy strategy to automatically search for (near-) optimal parameters of MLSS. Section 5.6 experimentally evaluates our solutions. Finally, we review related work in Section 5.7 and conclude in Section 5.8.

5.2 Problem Formulation and Background

5.2.1 Problem Formulation

We now formally define the problem of durability queries.

Stochastic Process. Consider a discrete time domain of interest $\mathbb{T} = \{1, 2, 3, \dots\}$ and a discrete-time stochastic process $\{v_t\}_{t \geqslant 0}$ with state space $\mathcal{X}$. For $\forall t \in \mathbb{T}$, $v_t = f(\mathcal{X}_t)$, where $f : \mathcal{X} \to \mathbb{R}$ is a deterministic value function. In general, the state $\mathcal{X}_t$ contains any necessary information for the computation of v_t; e.g., previous values/states and other auxiliary variables. The stochastic process develops in time according to probabilistic rules, i.e., conditional distributions of the form,

$$\mathbf{Pr}[\mathcal{X}_t \mid \mathcal{X}_{t-1}, \mathcal{X}_{t-2}, \dots, \mathcal{X}_{t-k}],$$

for integer $k \in [1, t-1]$. We showcase several common and concrete examples of stochastic process as follows.

1. Auto-Regressive Model or AR(p) model:

$$f(\mathcal{X}_t) = \sum_{i=1}^{p} \phi_i f(\mathcal{X}_{t-i}) + \epsilon_t,$$

   where ϕ and ϵ are model parameters, and p is the order of the model. The conditional distributions are explicitly controlled by these parameters.

2. Discrete-time Markov Chains is a special case of stochastic process that satisfies the Markov property:

$$\mathbf{Pr}[\mathcal{X}_t \mid \mathcal{X}_{t-1}, \mathcal{X}_{t-2}, \dots] = \mathbf{Pr}[\mathcal{X}_t \mid \mathcal{X}_{t-1}]\,,$$

   where the probability of each state depends only on previous state.

3. Recurrent Neural Networks:

$$\mathcal{X}_t \sim o\Bigg(g(h_{t-1}, \mathcal{X}_{t-1}; \theta); \theta\Bigg),$$

   where h_{t-1} is the state of the hidden layer(s) at time $t-1$, $o(\cdot)$ and $g(\cdot)$ are activation functions, and θ is the set of model parameters. The conditional distributions (RNN cell's output) are explicitly controlled by network's structure and trained parameters.

It is worth mentioning that throughout the chapter we generally do not assume any prior information on the stochastic process and its corresponding model; i.e., model parameters or transition probability matrix. The only interactions with the model that we require are simply simulations.

Durability Queries. Consider a time duration $t \in \mathbb{N}^+$, denote $SP = \{v_0, v_1, \dots, v_{t-1}\}$ as a realization of the probabilistic temporal series, called sample path, with length t by step-by-step simulations according to the stochastic process. Given a value threshold $\beta \in \mathbb{R}$, let $T_\beta = \inf\{t' \geqslant 0 \mid v_{t'} \geqslant \beta\}$; i.e., a random variable denoting the first time when the probabilistic temporal series reaches (or above) β. Given t and β, a durability query $Q(t, \beta)$ asks for the probability that a probabilistic temporal series crosses boundary of β before time t for the first time; i.e, $Q(t, \beta) := \mathbf{Pr}[T_\beta < t]$. In general, however, we cannot exactly answer the query (as we cannot even access the ground truth given a complex stochastic process[2]), but only estimate the quantity $\tau = \mathbf{Pr}[T_\beta < t]$ with certain quality guarantee; i.e., a tight confidence interval or small relative error. More specifically, our solution should output an estimator $\hat{\tau}$ for τ, together with a quality measurement. In real-life applications, the user could specify a quality requirement of the estimate and the algorithm should continuously update the result as time goes on, until it reaches the desired level. Alternatively, the user may also specify a time limit on the durability query processing, and the algorithm should return the best estimate obtained within the time limit, together with a quality measurement.

5.2.2 Background

Durability queries are deeply connected to a classic problem in statistic community, called First-Hitting Time or First-Passage Time in stochastic system [30, 90, 110]. Similar problems related to first-hitting time are also independently studied in very diverse fields, from economics [98] to ecology [40]. We briefly introduce several existing techniques for durability queries or generally for first-hitting time problem here, and lay the foundation for later sections.

Analytical Solution. As mentioned above, there exists exact analytical solutions for some basic and simple stochastic processes [51], e.g., Random Walks, AR(1) model, to name but a few. However, real applications often require more complex structures. For instance, Compound-Poisson process is a well known stochastic model for risk theory in financial worlds. In [112], authors derived an analytical solution for such stochastic processes. However, the exact solution itself is very complicated, involving multiple integrals that still require numerical approximations. In general, the analytical solution to first-hitting problem is model-specific and may not even exist

[2] For some simple probabilistic temporal series; e.g., an auto-regressive model, or AR(p), there exists analytical solution to answer the query $Q(t, \beta)$. But in general, we can only estimate the ground truth through simulations.

for most applications, thus cannot be directly used for durability query processing.

Simple Random Sampling (SRS). Answering durability queries through Monte Carlo simulations is the most general approach. Simple random sampling is the standard Monte Carlo technique. To answer durability query $Q(t, \beta)$, we randomly simulate n independent sample path $SP_i = \{v_0, v_1, \dots\}$ according to the underlying stochastic model. For each sample path, we define a label function indicating whether the first-hitting time of path SP_i is smaller than t:

$$l(SP_i) = \begin{cases} 1, & T_\beta(SP_i) < t \\ 0, & \text{otherwise} \end{cases}$$

where $T_\beta(SP_i)$ denotes the first-hitting time of sample path SP_i. Then, an unbiased estimator of SRS is

$$\hat{\tau}_{srs} = \frac{\sum_{i=1}^{n} l(SP_i)}{n}, \tag{5.1}$$

with estimated variance

$$\widehat{\text{Var}}(\hat{\tau}_{srs}) = \frac{\hat{\tau}_{srs}(1 - \hat{\tau}_{srs})}{n} \tag{5.2}$$

We use SRS as the main baseline solution throughout the chapter. The major drawback of SRS is the “blind search” nature – randomly simulates sample paths and just hopes that they could reach the target. For durability queries with small ground truth probability τ, SRS would waste significantly much simulation effort on those sample paths that never cross the value boundary.

Importance Sampling (IS). Importance sampling is one of the most popular variance reduction techniques for Monte Carlo simulations. It is a special case of biased sampling, where sampling distribution systematically differs from the underlying distribution in order to obtain more precise estimate using fewer samples. Let us use the following concrete example for better illustration. Consider an AR(1) model as the underlying stochastic process $f(\mathcal{X}_t) = c + \phi_1 f(\mathcal{X}_{t-1}) + \epsilon_t$, where c, ϕ_1 are constant parameter and ϵ_t is independent Gaussian noise, i.e., $\epsilon_t \sim N(0, \sigma)$ for $\forall t$. The transition probability for AR(1) model is controlled by noise distributions. Given the time threshold t, the random variable of interest $l(SP)$ has probability density $p(l) \sim \prod_{i=1}^{t} N(0, \sigma)$. SRS draws i.i.d samples from p and use Eq(5.1) as an unbiased estimation. By contrast, IS draws samples from an instrumental distribution q, and an unbiased estimator is

$$\hat{\tau}_{is} = \frac{1}{n} \sum_{i=1}^{n} \frac{p(l(SP_i))}{q(l(SP_i))} l(SP_i). \tag{5.3}$$

How to choose a good instrumental distribution q is critical for the success of IS. An iterative approach called Cross-Entropy(CE) method [35, 91] is widely used for importance sampling optimization. However, IS typically requires known information of the model as a priori; e.g., model parameters or state transition probabilities. This requirement can be impractical for some complex temporal processes, not to mention general black-box models that we consider in this chapter.

In general, all existing solutions as we reviewed above have its own limitations, and are not desirable for general durability query processing. In next section, we introduce a novel sampling approach that combines the best from both worlds – as general as SRS while as efficient as IS.

5.3 Multi-Level Splitting Sampling

Since generating too many paths that do not hit the value threshold can be a waste of simulation cost, it is natural to design a sampling procedure that more frequently produces paths that reach the target. For that reason, we introduce Multi-Level Splitting Sampling (MLSS) as an efficient alternative to existing solutions. Intuitively, MLSS creates an artificial drift of the simulations toward the target threshold.

Table 5.1: Notations

t	Time threshold of the simulation.
β	Value threshold of the simulation.
r	Splitting ratio; or number of copies.
m	Number of levels
SP	A sample path simulated from a stochastic temporal model.
L_i	The i-th level. L_m is the target level.
n_i	Number of first-time entrance state into L_i. n_0 is the number of root paths, n_m is the number of hits to the target.
p_i	(conditional) Level crossing probability from level L_{i-1} to L_i.

It rewards those sample paths that go in the "right" direction by splitting them into multiple copies. An important observation is that a path that has already reached β' (smaller than β but close to β) has higher probability to hit the target β, thus more simulation efforts should be spent on the promising partial paths instead of blindly starting a new path from scratch. It is worth mentioning that the idea of MLSS can be traced back to 1951 and has been studied by several authors in statistic community [68, 45]. Their discussions mainly focus on using MLSS on discrete-time Markov Chains with certain assumptions (will elaborate soon). For completeness, we introduce standard MLSS in the following section. The interested readers can refer to [72, 45] for a similar but more comprehensive introduction of MLSS. However, the standard MLSS has its restriction. In Section 5.4, we propose a novel and general form of MLSS that can be widely applied to any stochastic process.

Before diving into details of MLSS, we first introduce the basic idea of multi-level partition for estimating the probability τ. We then describe the sampling procedure, derive an unbiased estimator for MLSS, and finally present the variance analysis.

5.3.1 Multi-Level Partitioning

Assume the starting value for the temporal series is β_0 ($\beta_0 < \beta$). In the multi-level partitioning setting, we partition the interval $[\beta_0, \beta]$ into m disjoint sub-intervals with boundaries $\beta_0 < \beta_1 < \cdots < \beta_m = \beta$, effectively resulting in m levels where the i-th level $L_i = [\beta_i, \beta_{i+1})$ for $i < m$. The first level $L_0 = [\beta_0, \beta_1)$ is the starting level for all sample paths and the destination level is defined as $L_m = [\beta_m, +\infty)$. An important assumption for multi-level partitioning is that the probabilistic temporal series (more specifically, any sample path generated from the stochastic process) has to reach L_i before it reaches L_{i+1}, for $\forall i \leqslant m$. Formally, let $T_{L_i} = \inf\{t' \geqslant 0 \mid v_{t'} \in L_i\}$; i.e., a random variable denoting the first time that the series enters level L_i. And $\Xi_i = \{SP \mid T_{L_i}(SP) < t\}$; i.e., the set of all possible paths whose first hitting time to L_i is smaller than t. We assume that

$$\Xi_m \subset \Xi_{m-1} \subset \cdots \subset \Xi_1 \subset \Xi_0. \tag{5.4}$$

We refer to the above containment relationship as the "no level-skipping" assumption. In Section 5.4, we further discuss MLSS in a more general setting where there exists level-skipping.[3] Given the above assumption, we define the probability $p_i = \mathbf{Pr}[\Xi_i \mid \Xi_{i-1}]$ for $i > 1$, called level crossing probability. Then, we can decompose the target probability τ following the law of total probability,

$$\begin{aligned}\tau &= \mathbf{Pr}[\Xi_m] \\ &= \mathbf{Pr}[\Xi_m \mid \Xi_{m-1}] \cdots \mathbf{Pr}[\Xi_2 \mid \Xi_1]\,\mathbf{Pr}[\Xi_1 \mid \Xi_0]\,\mathbf{Pr}[\Xi_0] \\ &= \prod_{i=1}^{m} p_i.\end{aligned} \tag{5.5}$$

Overall, we try to decompose the event with probability τ (especially when τ is small) that hard to achieve into a series of more attainable objectives (with larger

[3] In discrete time, it is possible that a series jumps from a lower value to higher value crossing multiple levels between consecutive time instants.

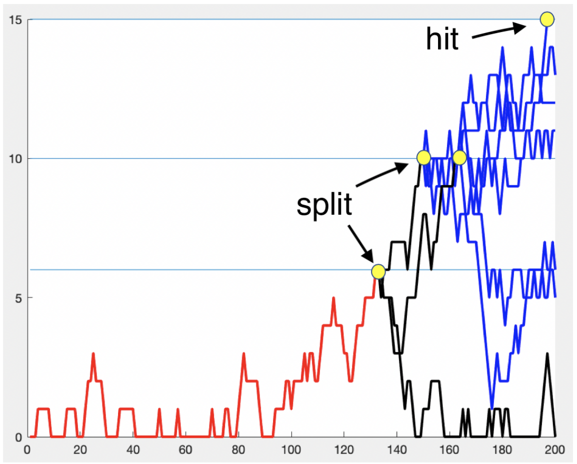


FIGURE 5.1: Simulation ($t = 200, \beta = 15$) of a root path using MLSS with splitting ratio $r = 3$.

level crossing probabilities p_i). We will show how such decomposability admits a new sampling approach in next sections.

5.3.2 Sampler and Estimator

MLSS Sampler. In a nutshell, MLSS works in rounds of stages, estimating the decomposed probability p_i "separately" between consecutive levels. For each level L_i, we maintain a counter n_i denoting the number of sample paths that first enter the level L_i. In the first stage, we start the simulation of a path from the initial level L_0, which we refer to as the "root path", and increment the counter n_0 by 1. We continue the simulation up to time t:

1. If the sample path does not enter the next level L_1, we stop and start a new round of simulation for the next root path.

2. Otherwise, we increment the counter n_1 by 1 and split the root path into r independent copies at the first time it enters L_1, where r is a constant called splitting ratio. Assume the hitting time is t', we define the state $\mathcal{X}_{t'}$ of sample path as the entrance state to L_1. All splitting copies from the original path

will use the same entrance state $\mathcal{X}_{t'}$ as starting point for future simulations.

Then in the next stage, for each of the splitting offspring of the root path, we recursively follow the similar procedure as described above: simulate the path up to time t; if it reaches the next level, increment the counter of that level, split and repeat; If not, finish the simulation at time t. The simulation of a root path stops when we finish the simulations of all its splitting offspring - either enters the target level L_m or runs until the time t. Figure 5.1 illustrates a concrete example of the simulations of one root path. Here we have $t = 200, \beta = 15$, a set of levels $L_0 = [0, 6), L_1 = [6, 10), L_2 = [10, 15), L_3 = [15, +\infty)$ and splitting ratio $r = 3$. The root path (red line) starts from L_0 and enters L_1 at timestamp 133. Then it splits into 3 copies (black lines) and continues the simulations forward. Two out of the three splitting paths (from L_1) enter L_2 and each of them further splits into three more copies (blue lines), respectively. Finally, one out of the six splits (from L_2) enters the target level L_3. All other copies (that do not have the chance to split) run till time t and stop. Following the above procedure, we sample and simulate n_0 root paths until the stopping criteria is met (will discuss in next section).

MLSS Estimator. Using the counters we have maintained through MLSS for each level, we have $\hat{p}_1 = \dfrac{n_1}{n_0}, \hat{p}_2 = \dfrac{n_2}{rn_1}, \ldots, \hat{p}_m = \dfrac{n_m}{rn_{m-1}}$. The estimator for MLSS is

$$\hat{\tau}_{mlss} = \prod_{i=1}^{m} \hat{p}_i = \frac{n_1}{n_0}\frac{n_2}{rn_1} \cdots \frac{n_m}{rn_{m-1}} = \frac{n_m}{n_0 r^{m-1}}. \tag{5.6}$$

Note that the $\hat{p}_i$'s are not necessarily independent. Despite the fact that we run independent simulations for each splitting copies of root paths, they share the same history of their common ancestor. But interestingly, we can prove that $\hat{\tau}_{mlss}$ is an unbiased estimator of τ, showing by the following proposition.

Proposition 14. *Under the "no level-skipping" assumption (5.4), using the Multi-Level Splitting Sampling with m levels and a splitting ratio r, $\hat{\tau}_{mlss}$ is an unbiased estimator of τ; that is, $\mathbb{E}[\hat{\tau}_{mlss}] = \tau$.*

Proof. Consider any level L_i, let S_i denote the set of entrance states of all possible paths that enter L_i, and S_i^j be a sample entrance state of S_i. Then, for each S_i^j, we can define a Bernoulli random variable variable $X_{i+1}(S_i^j) \sim Bernoulli(\Delta_{ij})$ with probability Δ_{ij}, denoting Bernoulli trials of a path (whether it reaches the next level L_{i+1} starting from state S_i^j in L_i). Note that Δ_i is a random variable with respect to the hitting probability from level L_i to L_{i+1}, and Δ_{ij} is a sample value (of Δ_i) based on entrance state S_i^j. An important observation is

$$\mathbb{E}[\Delta_i] = p_{i+1}, \tag{5.7}$$

that is, the expectation of success probability (of entering L_{i+1}) over all entrance states in L_i equals p_{i+1}, which is the probability that a path enters L_{i+1} conditioning on its entrance to L_i.

Now assume that during MLSS procedure, we have n_i entrance states in L_i: $S_i^1, S_i^2, \ldots, S_i^{n_i}$. For each of them we split and simulate r independent copies and watch whether they hit the next level L_{i+1}. Recall that the observations of each S_i^j is a Bernoulli variable with probability Δ_{ij}, thus the number of hits to L_{i+1} contributed by state S_i^j follows a binomial distribution as $B(r, \Delta_{ij})$. Combining n_i states together, we have the following formula of counter n_{i+1}:

$$n_{i+1} = \sum_{j=1}^{n_i} \sum_{k=1}^{r} X_{i+1}(S_i^j) \sim \sum_{j=1}^{n_i} B(r, \Delta_{ij}). \tag{5.8}$$

Since n_{i+1} conditions on n_i and the sampled entrance states, we further have

$$
\begin{aligned}
\mathbb{E}[n_{i+1} \mid n_i] &= \mathbb{E}\Bigg[\mathbb{E}[n_{i+1} \mid n_i, S_i^1, S_i^2, \ldots, S_i^{n_i}]\Bigg] \\
&\quad \text{(law of total expectation)} \\
&= \mathbb{E}\Bigg[\mathbb{E}[\sum_{j=1}^{n_i} B(r, \Delta_{ij}) \mid n_i, S_i^1, S_i^2, \ldots, S_i^{n_i}]\Bigg] \\
&\quad \text{(by Eq(5.8))} \\
&= \mathbb{E}[\sum_{j=1}^{n_i} r \cdot \Delta_{ij}] = n_i r \mathbb{E}[\Delta_i] \\
&= n_i r p_{i+1}. \quad \text{(by Eq(5.7))}
\end{aligned}
\tag{5.9}
$$

Applying this to $\hat{p}_{i+1}$ results in

$$
\mathbb{E}[\hat{p}_{i+1} \mid n_i] = \mathbb{E}[\frac{n_{i+1}}{rn_i} \mid n_i] = \frac{\mathbb{E}[n_{i+1} \mid n_i]}{rn_i} = \frac{n_i r p_{i+1}}{rn_i} = p_{i+1} \tag{5.10}
$$

Finally, unconditioning on n_i by law of total expectation,

$$
\mathbb{E}[\hat{p}_{i+1}] = \mathbb{E}\Bigg[\mathbb{E}[\hat{p}_{i+1} \mid n_i]\Bigg] = \mathbb{E}[p_{i+1}] = p_{i+1}. \tag{5.11}
$$

Next, we prove that $\mathbb{E}[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_m] = \mathbb{E}[p_1 p_2 \cdots p_m]$ by induction on m. First, we know that at the starting level all root paths are independently simulated, thus it is obvious that $\hat{p}_1$ is an unbiased estimator for p_1; that is, $\mathbb{E}[\hat{p}_1] = p_1$. Then, let us assume that $\mathbb{E}[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_{m-1}] = p_1 p_2 \cdots p_{m-1}$. Hence, we have

$$
\begin{aligned}
\mathbb{E}[\hat{\tau}_{mlss}] &= \mathbb{E}[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_m] \\
&= \mathbb{E}\Bigg[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_{m-1} \mathbb{E}[\hat{p}_m \mid n_{m-1}]\Bigg] \quad \text{(law of total expectation)} \\
&= \mathbb{E}[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_{m-1} p_m] \quad \text{(by Eq(5.9))} \\
&= \mathbb{E}[\hat{p}_1 \hat{p}_2 \cdots \hat{p}_{m-1}] p_m \\
&= p_1 p_2 \cdots p_m = \tau, \quad \text{(by assumption)}
\end{aligned}
\tag{5.12}
$$

which finishes the induction and proves that $\hat{\tau}_{mlss}$ is an unbiased estimation of τ. $\square$

Variance Analysis. Now let us derive the (estimated) variance of the MLSS estimator $\hat{\tau}_{mlss}$. Assume that we have sampled and simulated n_0 root paths, then the variance of our estimate is

$$\mathrm{Var}(\hat{\tau}_{mlss}) = \mathrm{Var}(\frac{n_m}{n_0 r^{m-1}}) = \frac{\mathrm{Var}(n_m)}{n_0^2 r^{2(m-1)}}. \tag{5.13}$$

Recall that root paths from the starting level L_0 are independent of each other. Since the number of hits to the target level (that is, n_m) all comes from the offspring of those n_0 root paths, we can further decompose the variance term into

$$\mathrm{Var}(n_m) = \sum_{i=1}^{n_0} \mathrm{Var}(\mathrm{Y_i}) = n_0 \mathrm{Var}(Y_1), \tag{5.14}$$

where Y_i denotes the number of hits to the final target contributed by the root path i and $\mathrm{Var}(Y_i) = \mathrm{Var}(Y_j)$ for any $i, j < n_0$. Finally, we have

$$\mathrm{Var}(\hat{\tau}_{mlss}) = \frac{\mathrm{Var}(Y_1)}{n_0 r^{2(m-1)}}. \tag{5.15}$$

It is hard to derive an analytical expression for $\mathrm{Var}(Y_1)$ in the multi-level splitting setting because there are many dependencies caused by splitting and sharing. However, we can easily estimate $\mathrm{Var}(Y_1)$ through the simulations using the standard variance estimator

$$s^2 = \frac{\sum_{i=1}^{n_0}(Y_i - \bar{Y})^2}{n_0 - 1}. \tag{5.16}$$

Combining it together, we have an unbiased estimation of the variance of MLSS estimator as follows:

$$\widehat{\mathrm{Var}}(\hat{\tau}_{mlss}) = \frac{s^2}{n_0 r^{2(m-1)}}. \tag{5.17}$$

5.3.3 *Relationship between SRS and MLSS.*

The following proposition shows the close connection between SRS and MLSS.

Proposition 15. *Simple random sampling is a special case of Multi-Level Splitting Sampling with splitting ratio $r = 1$.*

It is not hard to prove the above statement. As $r = 1$, $\hat{\tau}_{mlss} = \dfrac{n_m}{n_0} = \hat{\tau}_{srs}$. Similarly, $\mathrm{Var}(\hat{\tau}_{mlss}) = \dfrac{\mathrm{Var}(Y_1)}{n_0} = \dfrac{\tau_{srs}(1-\tau_{srs})}{n_0} = \mathrm{Var}(\tau_{srs})$.

Proposition 15 also gives us more insights on why MLSS could be a more efficient sampling procedure than SRS.

- Splitting Mechanism: SRS (or equivalently, MLSS with no splitting) always simulates a sample path up to time t. As discussed earlier, it could end up with a large portion of simulation cost wasted on paths that do not hit the boundary. In contrast, the splitting mechanism effectively makes the simulations more focused on those promising candidate paths. Though on average, MLSS has a higher per-root-path simulation cost than SRS, it potentially reduces the total number of root paths required, by producing more hits to the target.

- Multi-Level Partitioning: Another simulation-efficient feature of MLSS is the multi-level partitioning of the value threshold. Recall Eq(5.5) that τ can be decomposed into the product of a series of conditional probabilities by multi-level partitioning. Compared to τ, p_i's are larger probabilities. In other words, considering the simulations between levels, we create hits (either to the next level or to the final target) with higher probability, which further potentially avoids the waste of simulations on those non-hitting paths.

However, we still need careful considerations to apply MLSS in practice for the best performance; e.g., how to select splitting ratio r, how many partitions of levels

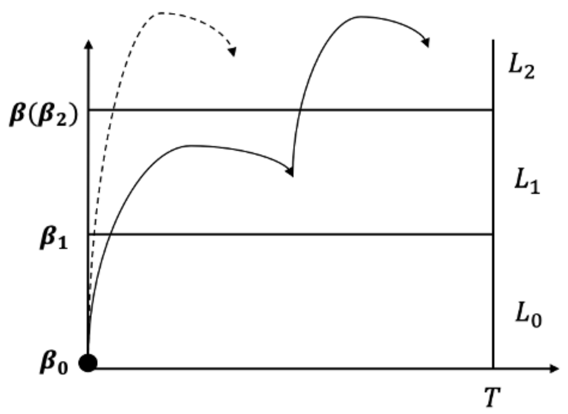


FIGURE 5.2: A simple two-level case with level-skipping. Dashed path represents a (discrete-time) series that directly goes from L_0 to L_2 without entering L_1.

do we need and how to decide the boundaries of partitions. There are many trade-offs among those choices. We discuss how to solve for the optimal setting of MLSS that minimizes the simulation cost in Section 5.5.

5.4 Extensions and Variants

In previous section, we elaborate the main idea and sampling techniques of the standard MLSS, and lay the theoretical foundations for its unbiased estimation (Proposition 14) and variance computation (Eq(5.15)) under the "no level-skipping" assumption (recall Eq(5.4)). However, given a discrete-time stochastic process, the assumption may not always be practical in real-life applications. A noisy and volatile temporal series with large value fluctuation between consecutive timestamps (e.g., stock prices) could easily break the assumption. Sometimes a bad level partitioning (e.g., level boundaries are placed too close to each other) also creates level-skipping. To go beyond the constraints and make MLSS widely applicable in practice, we present a novel and general MLSS procedure in the following section, and introduce some interesting variants of MLSS.

5.4.1 MLSS in General Form

For concreteness but without loss of generality, let us consider a simple case with two levels and with possible level-skipping, shown as Figure 5.2. There are two types of paths hitting the value boundary: solid line path (with no level-skipping) and dashed line path (skip from L_0 to L_2). With abuse of notation, let $p_{0,1} = p_1$ and $p_{1,2} = p_2$ (recall Eq(5.4) and Eq(5.5)). The numbers in subscript simply represent the transition between levels. These probabilities represents the normal case as we discussed in Section 5.3. However, with the existence of level-skipping, we need to introduce an additional probability $p_{0,2}$ denoting the chance of level-skipping. Hence, the ground truth hitting probability consists of two parts:

$$\tau = p_{0,1}p_{1,2} + p_{0,2}. \tag{5.18}$$

Sampler, Estimator and Variance. The sampling procedure is largely similar to standard MLSS as in Section 5.3: (1) path (recursively) splits into r copies the first time it enters into a higher level; and (2) maintain entrance counters n_0, n_1, n_2 for L_0, L_1, L_2. The only difference is that we decompose counter n_2 (number of hits to the target) as $n_2 = n_2^{(ns)} + n_2^{n(s)}$, where $n_2^{(ns)}$ is the number of hits from non-skipping paths while $n_2^{(s)}$ is the number of hits from level-skipping paths. Then, our estimator also consists of the estimations of these two parts,

$$\hat{\tau}_{mlss} = \frac{n_2^{(ns)}}{n_0 \cdot r} + \frac{n_2^{(s)}}{n_0} \tag{5.19}$$

Proposition 16. *In general, using the Multi-Level Splitting Sampling with 2 levels and a splitting r, $\hat{\tau}_{mlss}$ is an unbiased estimator of τ; that is $\mathbb{E}[\hat{\tau}_{mlss}] = \tau$.*

Proof.

$$\mathbb{E}[\hat{\tau}_{mlss}] = \mathbb{E}[\frac{n_2^{(ns)}}{n_0 \cdot r}] + \mathbb{E}[\frac{n_2^{(s)}}{n_0}]$$

In Proposition 14, we have proved that $\mathbb{E}[\frac{n_2^{(ns)}}{n_0 \cdot r}] = p_{0,1}p_{1,2} = p_1 p_2$. It is also clear that $\mathbb{E}[\frac{n_2^{(s)}}{n_0}] = p_{0,2}$. Thus, $\mathbb{E}[\hat{\tau}_{mlss}] = p_{0,1}p_{1,2} + p_{0,2} = \tau$. □

The variance of general MLSS is complicated and needs more care. Let us still focus on the simple two-level case. We have,

$$\text{Var}(\hat{\tau}_{mlss}) = \frac{\text{Var}\left(n_2^{(ns)}\right)}{n_0^2 r^2} + \frac{\text{Var}\left(n_2^{(s)}\right)}{n_0^2}.$$

First, $n_2^{(s)}$ can be viewed as a binomial variable with n_0 trials and probability $p_{0,2}$; i.e., $n_2^{(s)} \sim B(n_0, p_{0,2})$. Thus $\text{Var}\big(n_2^{(s)}\big) = n_0 p_{0,2}(1 - p_{0,2})$. Second, the quantity $n_2^{(ns)}$ conditions on the number of paths without skipping (n_1), which it is also an random variable throughout the sampling procedure. We cannot just break it up as in standard variance analysis. Instead, we should do a conditioning on number of non-skipping paths and use the law of total variance:

$$\begin{aligned}
\text{Var}\big(n_2^{(ns)}\big) &= \text{Var}\Big(\mathbb{E}[n_2^{(ns)} \mid n_1]\Big) + \mathbb{E}\Big[\text{Var}(n_2^{(ns)} \mid n_1)\Big] \\
&= \text{Var}(n_1 r p_{1,2}) + \mathbb{E}[n_1 \text{Var}(Y_1)] \\
&\quad \text{(Recall Eq(5.9) and Eq(5.14))} \\
&= r^2 p_{1,2}^2 \text{Var}(n_1) + \mathbb{E}[n_1]\text{Var}(Y_1) \\
&= r^2 p_{1,2}^2 n_0 p_{0,1}(1 - p_{0,1}) + \mathbb{E}[n_1]\text{Var}(Y_1). \\
&\quad (n_1 \sim B(n_0, p_{0,1}))
\end{aligned}$$

Hence,

$$\frac{\text{Var}\big(n_2^{(ns)}\big)}{n_0^2 r^2} = p_{1,2}^2 \frac{p_{0,1}(1 - p_{0,1})}{n_0} + p_{0,1} \frac{\text{Var}(Y_1)}{n_0 r^2} \tag{5.20}$$

Putting it all together, we have

$$\begin{aligned}&\mathrm{Var}(\hat{\tau}_{mlss})\\&= p_{1,2}^2 \frac{p_{0,1}(1-p_{0,1})}{n_0} + p_{0,1}\frac{\mathrm{Var}(Y_1)}{n_0 r^2} + \frac{p_{0,2}(1-p_{0,2})}{n_0}.\end{aligned} \tag{5.21}$$

In practice, we can use $\hat{p}_{0,1} = \frac{n_1}{n_0}$ as an unbiased estimation for $p_{0,1}$. Similarly, $\hat{p}_{0,2} = \frac{n_2^{(s)}}{n_0}$ and $\hat{p}_{1,2} = \frac{n_2^{(ns)}}{n_1 r}$ as unbiased estimations for $p_{0,2}$ and $p_{1,2}$, respectively. $\mathrm{Var}(Y_1)$ can be estimated by s^2 as Eq(5.16) suggests. Again, it is not hard to find that the variance term we derived in Eq(5.15) is a special case of the above equation when there is no level-skipping, i.e., $p_{0,2} = 0$ and $p_{0,1} = 1$.

General MLSS with m Levels. Now let us move to a more general setting that there are m levels with possible level-skippings. Let $G = (V, E, \psi)$ be a directed graph representing the transitions among m levels for a stochastic process, where $V = \{L_0, L_1, \ldots, L_m\}$, $E = \{(L_i, L_j) \mid \forall i < j \leqslant m\}$ and ψ denotes the set of all possible paths between any two vertices in G. A path is a sequence of edges $(e_0, e_1, \ldots, e_{k-1}) \subseteq E$ for which there is a sequence of vertices $(L_0, L_1, \ldots, L_k) \subseteq V$ such that $e_i = (L_i, L_{i+1})$ for $i = 0, 1, \ldots, k-1$. Specifically, we are more interested in a subset of paths starting from vertex L_0 and ending at vertex L_m, denoted by $\psi(L_0, L_m) \subseteq \psi$. Then, the durability probability can be written as the following:

$$\tau = \sum_{\psi_i \in \psi(L_0, L_m)} \mathbf{Pr}[\psi_i], \tag{5.22}$$

where $\mathbf{Pr}[\psi_i]$ is the probability that the temporal process follows the level transitions by path ψ_i and hits the target. As for estimator, we again use the counters collected from each level during the sampling procedure, resulting in the following unbiased estimator for τ:

$$\hat{\tau}_{mlss} = \sum_{\psi_i \in \psi(L_0, L_m)} \frac{n_m^{(\psi_i)}}{n_0 \cdot r^{|\psi_i|-1}}, \tag{5.23}$$

where $n_m^{(\psi_i)}$ is the number of hits to the target contributed by sample paths that follows the level transitions of ψ_i.

Proposition 17. *In general, using the Multi-Level Splitting Sampling with m levels and a splitting ratio r, $\hat{\tau}_{mlss}$ is an unbiased estimator of τ; that is, $\mathbb{E}[\hat{\tau}_{mlss}] = \tau$.*

The proof can be straightforwardly derived from the proofs of Proposition 14 and Proposition 16, thus omitted.

The size of possible paths from L_0 to L_m, $\psi(L_0, L_m)$, is at most 2^{m-1}. We can see that our discussions in Section 5.3 with "no level-skipping" assumption is just a special case of the general MLSS, 1 out of 2^{m-1} possible cases. It is very hard to deliver analytic expressions for variance term in general m levels MLSS because of the dependencies among splitting paths due to the sharing of their history. But our discussions in Section 5.4.1 (from Eq(5.19) to Eq(5.21))) already showcase the variance derivations for a simple two-level case. The variance analysis for using m levels will be very complex but similar to the two-level case. We leave this part as one of the future work.

Though MLSS in general form still provides an unbiased estimation (recall Proposition 17), the variance of MLSS estimator becomes way more complicated, e.g., Eq(5.21), resulting in the decrease of its sampling efficiency. Not only is there variance caused by number of hits to the target among different root paths, but a multinomial variance due to the different level transition paths (up to 2^{m-1}) from L_0 to L_m. The latter variance could be potentially very large because of the inherent uncertain nature of stochastic process, especially when there are many level transition possibilities (i.e., a very volatile temporal process). Hence, in the very general case, MLSS may not have a clear benefit over SRS, or might be even worse than SRS in some cases. We recommend that the best working scenarios for MLSS should be the case where there is no level-skipping, as in Section 5.3. It is better to first analyse the temporal data (and its underlying stochastic process) that we are working with. If the process itself is very noisy and fluctuate much over time, then the standard SRS might be a safer choice.

5.4.2 Variants of MLSS

The idea of Multi-Level Splitting Sampling is general, and can be instantiated into different but similar forms. In this paper, we focus on the instance of MLSS (Section 5.3 and Section 5.4) that is more suitable for durability query processing. The interested readers can refer to [72] and [45] for a more comprehensive perspective on splitting-based sampling approach. In this section, we briefly introduce two interesting variants of MLSS.

Splitting Ratio. In our implementation of MLSS, we assume the splitting ratio r is a pre-defined constant number. However, this is not a hard constraint. Users can pick different number of splitting ratios for different levels during the sampling procedure, and we still have an unbiased estimator as the following:

$$\hat{\tau}_{mlss} = \frac{n_m}{n_0 \prod_{i=2}^{m} r_i}, \tag{5.24}$$

where r_i is the splitting ratio for level L_i. The corresponding variance expression is,

$$\mathrm{Var}(\hat{\tau}_{mlss}) = \frac{\mathrm{Var}(Y_1)}{n_0 \prod_{i=2}^{m} r_i^2}. \tag{5.25}$$

Furthermore, splitting ratios can be adjusted or learned throughout the sampling procedure on-the-fly as we gain more knowledge based on previous simulations, enabling an adaptive version of MLSS for potentially more efficient and robust performances. Similar ideas have been successfully applied to importance sampling for significant efficiency improvement [79]. How to design an efficient adaptive MLSS remains an interesting and challenging future work.

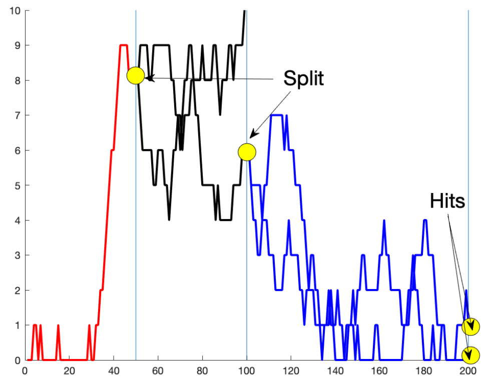


FIGURE 5.3: Simulation ($t = 200, \beta = 15$) of a root path using MLSS based on level partitions in temporal domain.

Level Partitions. As in Section 5.3.1, we formally introduce the notion of multi-level partitioning in the value domain. Again, the definition of "levels" can also be general. For instance, in Figure 5.3, we illustrate the simulation of a root path using MLSS (r=3) based on level partitions in temporal domain. The MLSS method, together with its sampler, estimator and variance analysis, are largely independent on users' choice of level partitions. Mathematically speaking, the notion of levels is simply a set of conditions determining when a sample path should be splitted.

Nonetheless, on the other hand, the choice of levels is critical for the overall efficiency of MLSS, since it directly controls the "incentive mechanism". The definition of levels should reward and lead simulations towards the "right" directions in order to improve sampling efficiency. Users can pick the more suitable version of levels based on their temporal models, applications and queries.

5.5 Optimizations

There is still one missing piece of MLSS: how to choose parameters? More specifically, how many levels do we need, and given the number of levels, how to properly divide level partitions? In this section, we first present an empirical evaluation metric for MLSS with different parameter settings, and then introduce an adaptive greedy partition strategy that automatically searches for the (near-) optimal parameters of MLSS, i.e., number of levels and level partitions.[4]

5.5.1 Partition Plan Evaluation

From previous studies in statistic community, authors [72] showed an analogy between MLSS (with fixed splitting ratio) and branching process theory [53], and concluded that the optimal setting for MLSS is to make crossing probabilities between consecutive levels roughly the same, called a "balanced growth." That is, consider MLSS with m levels,

$$p_1 = p_2 = \cdots = p_m = p = \tau^{1/m}. \tag{5.26}$$

Under this setting, from standard branching process theory, we have

$$\mathrm{Var}(\hat{\tau}_{mlss}) = \frac{m(1-p)p^{2m-1}}{n_0}. \tag{5.27}$$

The above expression indicates that given the fixed number of root paths, more levels lead to smaller variance. However, in the meantime, more levels lead to more expensive simulation cost of a root path because of the exponential splitting growth of a root path through the levels. Our optimization goal, using MLSS as approximate query processing technique, is not just to minimize variance. Instead, we hope to minimize the variance in a fixed amount of the time. Since, ultimately, the query time using MLSS is determined by the variance of the estimator. A smaller variance in unit time directly leads to less simulation cost for answering durability query. Though the "balanced growth" strategy is a reasonable guideline to partition the space, it is still not clear how to choose the right number of levels.

[4] For simplicity and solution efficiency, we only consider MLSS with no level-skipping assumption (recall discussions in Section 5.3).

To meet the new criteria, we present the following evaluation metric. Consider a level partition plan s, which consists of a set of values what we call "partition boundaries"; that is, $s = \{v \mid v \in (\beta_0, \beta)\}$. Given a fixed amount of simulation budget, say t time, we have simulated n_t root paths (including all its splitting copies). Note that n_t is a random variable depending on the total time t and the average simulation time c_s of a root path using partition plan s. We define an evaluation function for s in terms of variance of estimator $\hat{\tau}_{mlss}$ by n_t:

$$eval(s) = \mathrm{Var}(\frac{n_m^{(t)}}{n_t r^{m-1}}), \tag{5.28}$$

where $n_m^{(t)}$ denotes the total number of target hits within t time. In this case, $m = |s| + 1$, denoting the total number levels induced by plan s. Since n_t is a random variable, we should express the variance term conditioning on n_t, and use the law of total variance and the decomposition trick in Eq(5.15):

$$eval(s) = \mathbb{E}\left[\frac{\mathrm{Var}(Y_1)}{n_t r^{2(m-1)}} \mid n_t\right] + \mathrm{Var}\left(\mathbb{E}[\frac{n_m^{(t)}}{n_t r^{m-1}} \mid n_t]\right) \tag{5.29}$$

Given n_t, $\mathbb{E}[\frac{n_m^{(t)}}{n_t r^{m-1}}] = \tau$, thus the second term in the above equation is 0. Recall that Y_1 is a random variable denoting the number of target hits from a root path. Given n_t and partition plan s, $\mathrm{Var}(Y_1)$ becomes a constant. Hence, the first term in the equation becomes $\frac{\mathrm{Var}(Y_1)}{r^{2(m-1)}}\mathbb{E}[1/n_t]$. The term $\mathbb{E}[1/n_t]$ can be roughly estimated by $1/\frac{t}{c_s} = c_s/t$. Finally, the evaluation function of a partition plan s is

$$eval(s) = \frac{\mathrm{Var}(Y_1)}{r^{2(m-1)}}\frac{c_s}{t}. \tag{5.30}$$

Ideally, given a fixed amount of time t, we hope to solve for partition plan s that minimizes the objective $eval(s)$. However, this optimization problem is hardly solvable analytically, since $\text{Var}(Y_1)$ and c_s are themselves variables when the plan s changes. But fortunately, we can optimize the objective empirically, as $\text{Var}(Y_1)$ and c_s can be estimated through the MLSS simulations. Eq(5.16) provides an unbiased estimator for $\text{Var}(Y_1)$, and c_s can also be estimated simply dividing t by the number of simulations of root path within time t. In this way, we can start with a candidate pool of partition plans. For each candidate, we run MLSS simulations for the same amount of time t as trial runs to estimate the objective $eval(s)$, and finally pick the best candidate who produces the minimum value.

5.5.2 An Adaptive Greedy Partition Strategy

To empirically optimize MLSS, it is prohibitively expensive to run trial simulations for all feasible partition plans and splitting ratios. In this section, we present a heuristic greedy strategy that works well in practice to automatically search for (near-) optimal MLSS parameters.

The main idea of our strategy is to adaptively and recursively partition the space – place the partition boundaries one by one and always partition the level with smaller level crossing probability. The intuition behind our greedy behavior is two-fold: (1) A level with smaller level crossing probability means that this level is the "obstacle" blocking sample paths reaching the target. Partition such levels would focus the simulation resources more on success paths; (2) as we recursively bisect levels with smaller crossing probability, it automatically moves towards a "balanced growth" situation where crossing probabilities from all levels are roughly the same. Recall our discussion in Section 5.5.1; this behavior has already been confirmed by [72] to have a better sampling efficiency.

Full description of the algorithm is shown in Algorithm 5. Throughout the procedure, we adaptively make two decisions: what is the optimal number of levels, and how to optimally produce such number of levels? At the beginning, we start with the

Algorithm 5: Adaptive Greedy Partition.

```
Input  : A value interval $I = [\beta_0, \beta]$.
Output: A partition boundary set $B = \{v \mid v \in [\beta_0, \beta]\}$.
1  $B \leftarrow \varnothing$;
2  $opt\_eval \leftarrow INT\_MAX$ //remember the minimum evaluation value so far;
3  $v_{lo} \leftarrow \beta_0, v_{hi} \leftarrow \beta$;
4  for $round\ i = \{1, 2, \cdots\}$ do
5      Uniformly generate a value set $C = \{v \mid v \in (v_{lo}, v_{hi})\}$ as candidates for
         the $i$-th partition boundary;
6      $e^* = \min_{v \in C} eval(B \cup v)$;
7      $v^* = \arg\min_{v \in C} eval(B \cup v)$;
8      if $e^* < opt\_eval$ then
9          $B \leftarrow B \cup v^*$;
10         $opt\_eval \leftarrow e^*$;
11         Find the level $[\beta_i, \beta_j](\beta_i, \beta_j \in B, \beta_i < \beta_j)$, induced by $B$ and $I$, that
             has the smallest level crossing probability $p_{i,j}$;
12         $v_{lo} \leftarrow \beta_i, v_{hi} \leftarrow \beta_j$;
13     else
14         break;
15 return $B$;
```

original interval $[\beta_0, \beta]$ (Line 1). Then we place the partition boundary one by one, recursively bisecting the value intervals, until a stopping condition is met (Line 4-14). In the loop, we first generate a set of candidate boundaries (Line 5) and then use the empirical evaluation approach, as elaborated in Section 5.5.1, to find the optimal partition boundary (Line 6 and Line 7). Finally, we need to update our partition plan and decide when to stop the procedure. If the current best evaluation is better the previous, we continue to add a new partition boundary (Line 8-12). Note that here we need to greedily pick the next interval with smallest crossing probability to partition (Line 11-12). Otherwise, if the current best evaluation is already worse than the previous, there is no need to further add more partition boundaries to the plan, since more levels lead to exponential growth of splitting paths and would incur more expensive simulation cost overall.

Given the aforementioned empirical optimization framework, it saves users from the time-consuming manual parameters tuning process when applying MLSS in practice. Users only need to pick a reasonable splitting ratio (will further justify the choice of splitting ratio in Section 5.6.3) in advance, and our optimization framework will take care of the rest. Interestingly, even though the splitting ratio is a pre-defined constant, our optimization strategy also implicitly allows tuning the splitting ratio as well by exploring different number of levels, since a level with a larger splitting ratio can be approximated by multiple levels with the constant splitting ratio.

We conclude this section with a rough complexity analysis of our greedy strategy. Assume our algorithm runs up to m rounds and for each round we generate $|C|$ candidates. Overall, we need $O(m|C|)$ number of partition plans to try and evaluate. Compared to the baseline solution that tries all possible partition plans for each number of levels, which is up to $(|C|^m)$ plans in total, the greedy solution provides a similar quality of plan, but with significant efficiency improvement. An additional benefit of our empirical optimization solution is that all the trial runs of MLSS are not "wasted." Since each trial simulation, no matter which plan it follows, returns an unbiased estimator. So in the meantime we are picking the optimal parameters, we are also building up towards a reliable estimation for the query.

5.6 Experiments

In this section, we comprehensively and quantitatively evaluate the performance of our proposed solutions. Section 5.6.1 elaborates the experiment setup, including datasets, evaluation metrics and implementation details. Section 5.6.2 presents an overall comparison between the proposed solution MLSS and the standard Monte Carlo technique SRS on a variety types of durability queries with different underlying stochastic processes. Finally in Section 5.6.3, we focus on fine-tuning the MLSS procedure, and evaluates the proposed empirical optimization framework.

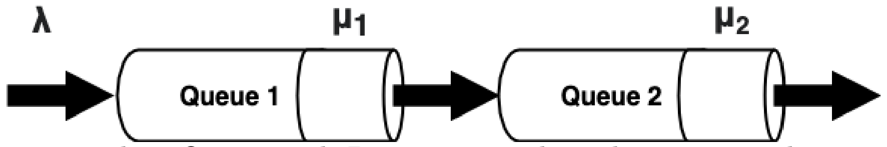


FIGURE 5.4: Tandem Queue with Poisson arrivals and Exponential service time.

5.6.1 Experiment Setup

Stochastic Processes and Datasets. We select three stochastic temporal processes that are commonly used in practical applications.

1. **Tandem Queues**: As shown in Figure 5.4, we have a queueing systems with tandem queues, which is the simplest non-trivial network of queues in queueing theory [28]. The process is the following. Customers come into Queue 1 following a Poisson distribution with $Pois(\lambda)$. Queue 1 services each customer following an Exponential distribution with $Exp(\mu_1)$, and then sends customers into Queue 2. Queue 2 services each customer by another Exponential distribution with $Exp(\mu_2)$ before customers leave the system. We consider the number of customers in Queue 2 as a stochastic process, and always start with an empty system (i.e., two empty queues). In our experiments, we set $\lambda = 0.5, \mu_1 = \mu_2 = 2$. Though in simple form, queueing system is the foundation for many real world problems [84], to name a few, birth-death process, supply chains, transportation scheduling and computer networks analysis [71].

2. **Compound-Poisson Process:** A Compound-Poisson Process (CPP) can be described by the following stochastic process $U = \{U(t)\}_{t\geqslant 0}$:

$$U(t) = u + ct - S(t), \tag{5.31}$$

where $S(t)$ is a compound Poisson process with jump density λ and jump distribution F, and $u, c > 0$ are constants. This type of process is commonly used

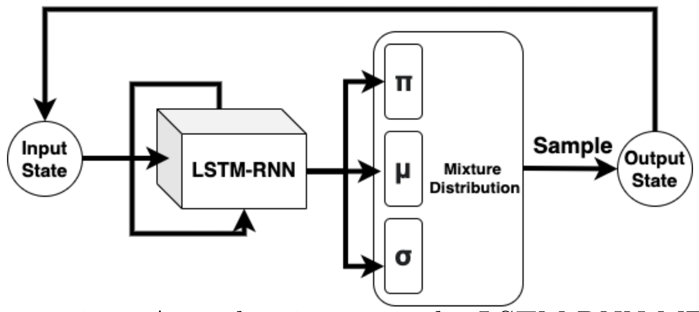


FIGURE 5.5: A stochastic process by LSTM-RNN-MDN.

in financial world for risk management and financial product design [10]. Intuitively, imagine a insurance policy with u as initial surplus and c as users' monthly payment. The compound Poisson process $S(t)$ represents the aggregate claim payments up to time t. Then the overall stochastic process U shows the net profit of this insurance policy. In our experiments, we set Poisson jump density $\lambda = 0.8$ and use uniform distribution $Uni(5, 10)$ as jump distribution. For constants, $u = 15$ and $c = 4.5$.

3. **Recurrent Neural Networks:** As shown in Figure 5.5, we train a Recurrent Neural Networks (RNN) with Long-Short Term Memory (LSTM) and Mixture Density Network (MDN) [15] using Google's 5-year daily stock prices from 2015 to 2020. The LSTM-RNN-MDN structure has proven its success on many real life tasks of probabilistically modelling and generating sequence data, for instance, language models [12], speech recognition[49], hard-writings analysis[50] and music composition [36]. In our network, we use two stacked RNN layers, 256 LSTM units per RNN layer, and a 2-dimensional mixture layer with 5 mixtures. During training phase, we trained the model for sequence length of 50, and for 100 epochs with a batch size of 32.

Evaluation Metric. Performance of different methods are evaluated using the following two metrics: total number of simulation steps (recall that a sample path is simulated step-by-step using a stochastic process) and total simulation time. In our experiments, we run sampling procedures until the estimation satisfies certain quality measurement. Specifically, we use two quality measurements throughout the experiment section:

1. **Confidence Interval:** Confidence interval (CI) is a statistical measurement for point estimate. It shows how likely (or how confident) that the true parameter is in the proposed range. There is no universal formula to construct CI for an arbitrary estimator. However, if a point estimator $\hat{\mu}$ takes the form of the mean of n independent and identically distributed (i.i.d.) random variables with equal expectation μ, then by the Central Limit Theorem and Normal Approximation, an approximate 1-α CI of μ can be constructed by:

$$[\hat{\mu} - z_{\alpha/2}\sqrt{\frac{\sigma^2}{n}}, \hat{\mu} + z_{\alpha/2}\sqrt{\frac{\sigma^2}{n}}], \tag{5.32}$$

   where $z_{\alpha/2}$ is the Normal critical value with right-tail probability $\alpha/2$, and σ^2 is the variance of estimate. By default, to obtain reliable query answers, we require that all estimations should have a 1% CI with 95% confidence level (i.e., $z_{\alpha/2} = 1.96$). Unfortunately, the standard CI as in the above equation has its limitation. When the true probability μ is very close to 0 or 1, where the Normal Approximation assumption does not hold, the CI guarantee would break.

2. **Relative Error:** Relative Error (RE) measures the variance (of estimate) as a relative ratio to the true probability, defined as follows:

$$RE = \frac{\sqrt{\sigma^2}}{\mu}, \tag{5.33}$$

   where μ is the true probability and σ^2 is the variance of estimate. This is not feasible to calculate directly in practice, since we do not know the true probability

Table 5.2: Query settings on different models.

Query Type	Medium	Small	Tiny	Rare
Queue Model	$t : 500$ $\beta : 20$	$t : 500$ $\beta : 26$	$t : 500$ $\beta : 40$	$t : 500$ $\beta : 45$
CPP Model	$t : 500$ $\beta : 300$	$t : 500$ $\beta : 350$	$t : 500$ $\beta : 450$	$t : 500$ $\beta : 500$
RNN Model	-	$t : 200$ $\beta : 1550$	$t : 200$ $\beta : 1600$	-

μ before the query. But in practice, we can roughly estimate the ground truth probability, and use that to fairly compare the RE ratio among different methods. Unlike CI, RE is widely applicable in any scenarios.

In sum, throughout the experiment section, we evaluate durability queries with different ground truth probabilities. For queries that have small-to-moderate probability (i.e., > 0.05), we use Confidence Interval as the quality measure. For queries that have tiny probability (i.e., 10^{-4} to 10^{-2}), we use Relative Error as the alternative measure.

Implementations. All stochastic temporal models and proposed solutions were implemented in Python3. More specifically, for neural network's construction and training, we use Keras [26] (back end by TensorFlow [2]). For MLSS, we use splitting ratio $r = 3$ and a "balanced-growth" level-partition plan (recall discussions in Section 5.5.1) as the default experiment setting. For simplicity and efficiency considerations, we make sure that the selected stochastic processes and our level partitions will not incur "level skipping." Thus, the MLSS sampler, estimator and variance formula used in all experiments are the ones describe in Section 5.3. All experiments were performed on a Linux machine with two Intel Xeon E5-2640 v4 2.4GHz processor with 256GB of memory.

Table 5.3: Query answer comparisons on Queue Model. Results are averaged over 100 runs with standard deviation.

Query Type	Medium	Small	Tiny	Rare
SRS	17.2% $\pm$ 0.5%	5.1% $\pm$ 0.5%	0.15% $\pm$ 0.03%	0.04% $\pm 2e^{-5}$
MLSS	17.9% $\pm$ 0.4%	5.5% $\pm$ 0.5%	0.17% $\pm$ 0.02%	0.04% $\pm 3e^{-5}$

Table 5.4: Query answer comparisons on CPP Model. Results are averaged over 100 runs with standard deviation.

Query Type	Medium	Small	Tiny	Rare
SRS	15.5% $\pm$ 0.5%	5.3% $\pm$ 0.5%	0.24% $\pm$ 0.02%	0.03% $\pm 3e^{-5}$
MLSS	15.6% $\pm$ 0.4%	5.3% $\pm$ 0.5%	0.26% $\pm$ 0.01%	0.03% $\pm 4e^{-5}$

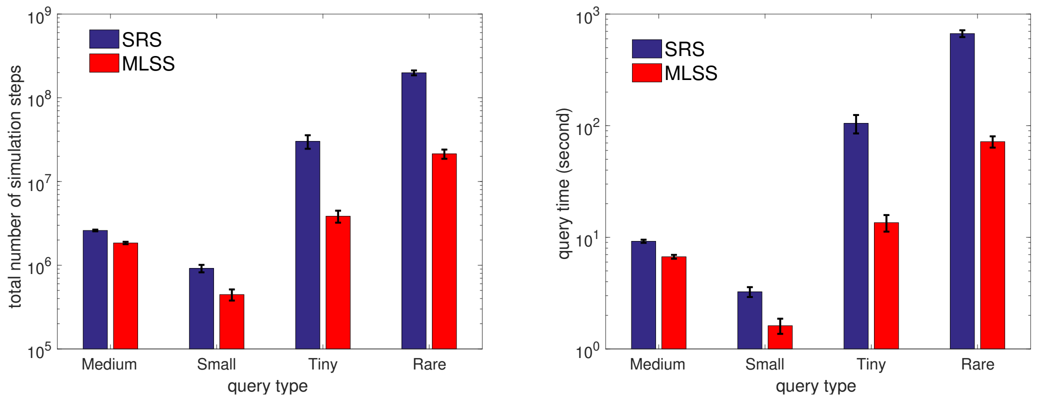
5.6.2 MLSS vs. SRS

In this section, we comprehensively compare the performance of MLSS and SRS. For each stochastic temporal model, we design four types of durability queries: Medium, Small, Tiny and Rare, denoting the quantity of the (estimated) true answer probability of the queries. Detailed query parameters are summarized in Table 5.2.

Estimations and Overall Efficiency. We first demonstrate the answer quality, i.e., unbiasedness, of MLSS. For each model and for each type of query, we repeatedly run SRS and MLSS 100 times, respectively, and average the returned answers along with empirical standard deviations. Results are summarized in Table 5.3 (Queue

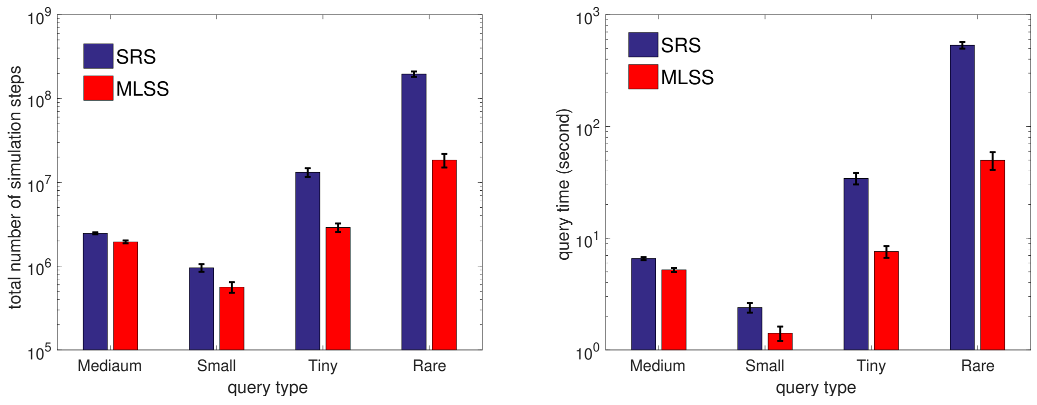
Table 5.5: Query performance (single run) comparisons on RNN Model.

Query Type	Small	Tiny
SRS	2.6% 1,009,431 steps 3.8 hours	0.51% 7,262,735 steps 33.7 hours
MLSS	1.9% 196,913 steps 0.75 hour	0.45% 804,035 steps 3.9 hours



(a) Simulation steps (b) Query time

FIGURE 5.6: Query efficiency on Queue Model

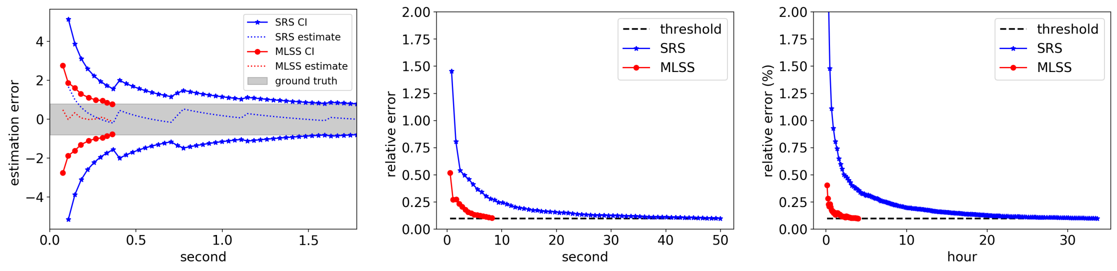


(a) Simulation steps (b) Query time

FIGURE 5.7: Query efficiency on CPP Model

Model), Table 5.4 (CPP Model) and Table 5.5 (RNN Model). As shown in these tables, the answers returned by SRS and MLSS, on all types of queries and on all models, are essentially the same. Even though they are not exactly the same, the differences are well contained by the standard deviation. This findings confirmed our analysis and proof in Section 5.3 about MLSS's unbiased estimation.

Next, let us compare the query efficiency between SRS and MLSS. We time the query until its answer (estimation) achieves certain quality measurement. For Medium and Small queries, we require the final estimation has a 1% confidence



(a) Queue Model, Small Query, CI (b) CPP Model, Tiny Query, RE (c) RNN Model, Tiny Query, RE

FIGURE 5.8: Query answer quality over time.

interval with at least 95% confidence. For Tiny and Rare queries, we require the final estimation has a 10% relative error (compared to the true probability). As shown in Figure 5.6 (Queue Model) and Figure 5.7 (CPP Model), MLSS generally runs significant faster than SRS (note the log scale on y-axis). For Medium and Small queries, we can see a 40% to 60% query time reduction brought by MLSS. For Tiny and Rare queries, MLSS runs 10x faster than SRS, without loss of answer quality. As discussed earlier in Section 5.3.3, the main advantage of MLSS to SRS is the ability to focus and encourage simulations that move towards the target. This property is especially helpful for those durability queries with lower probability, since MLSS can better distribute simulation efforts to promising paths hitting the target, instead of blindly wasting time on those failure paths (which would be a large portion of the total) as SRS did. Again, results in Figure 5.6 and Figure 5.7 are averaged over 100 runs. Standard deviations are shown as error bars on top of the bar charts. We observe similar query efficiency improvement on the more complex RNN model. In Table 5.5, for Small and Tiny queries (which are more commonly asked in practice) on RNN model, there is a roughly 80% to an order-of-magnitude query time reduction provided by MLSS.

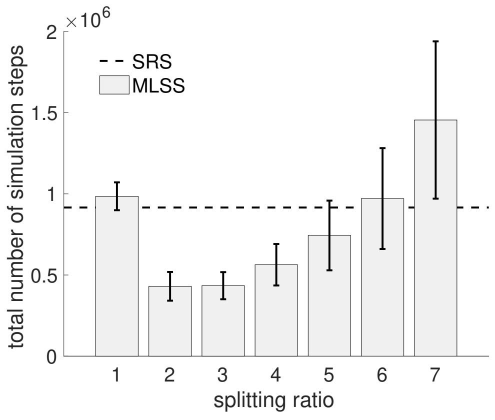
Overall, we can see that MLSS clearly surpasses SRS across different models and on different types of commonly asked durability queries in practice, providing query speedup from 40% up to an order-of-magnitude, without sacrificing answer quality.

Query Performance over Time. To take a closer look at the query performance comparison of MLSS and SRS, we monitor the query answers and its quality (CI or RE) over time, and plot the convergence of estimations on single run of MLSS and SRS, respectively. See Figure 5.8 for details. In Figure 5.8-(1), we run a Small query on Queue model and use CI as estimation quality measure. For better illustration, CI intervals are interpreted as a percentage to the true probability such that it will be centered at 0. The grey ribbon in the plot shows the desired region for a reliable estimate (true probability with 1% CI). Symmetric red lines and blue lines demonstrate how the CIs of MLSS and SRS converge over time, respectively. Red dotted line and blue dotted line are the estimate of MLSS and SRS over time. It is clear that MLSS converges faster than SRS on estimation quality, which is the main reason why MLSS is a more efficient sampling procedure or query processing technique than SRS. On the other hand, we can also see that the estimates (red dotted line and blue dotted line) from MLSS and SRS are always nicely contained by its corresponding CI, showing the statistical guarantees brought by CI. We observe similar behaviors on CPP model (Figure 5.8-(2)) and RNN model (Figure 5.8-(3)). Here we use run Tiny queries on these two models and use RE as quality measure. Similarly, the time that MLSS needs for a reliable estimate (10% RE, dashed line in the plot) is significantly shorter than that of SRS.

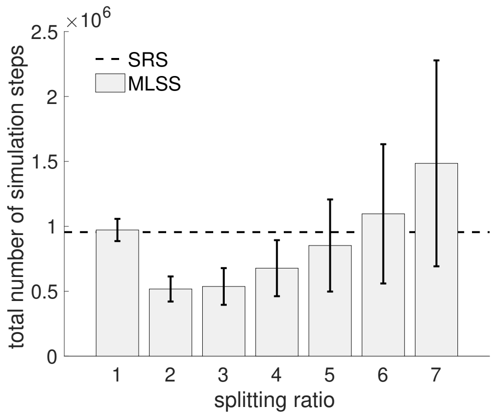
In sum, Figure 5.8 demonstrates the fast convergence of MLSS's sampler and estimator, which further explains the reason why MLSS can be notably efficient than SRS in general.

5.6.3 MLSS Optimization

In previous sections, we have shown the dominance of MLSS over SRS across a variety of models and query types. We now focus more on MLSS itself, and investigate

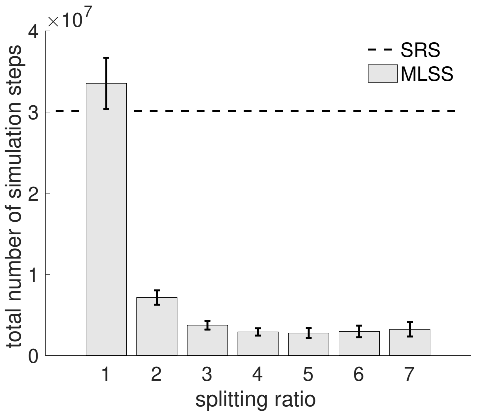


(a) Queue Model

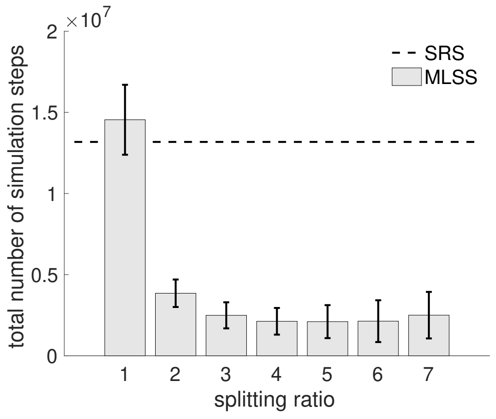


(b) CPP Model

FIGURE 5.9: Trade-off between splitting ratio and MLSS's overall efficiency on Small Query.



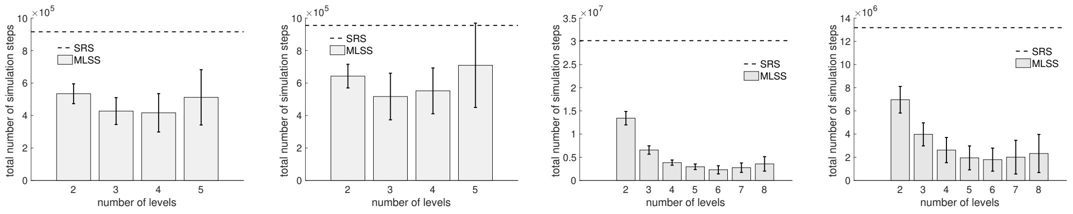
(a) Queue Model



(b) CPP Model

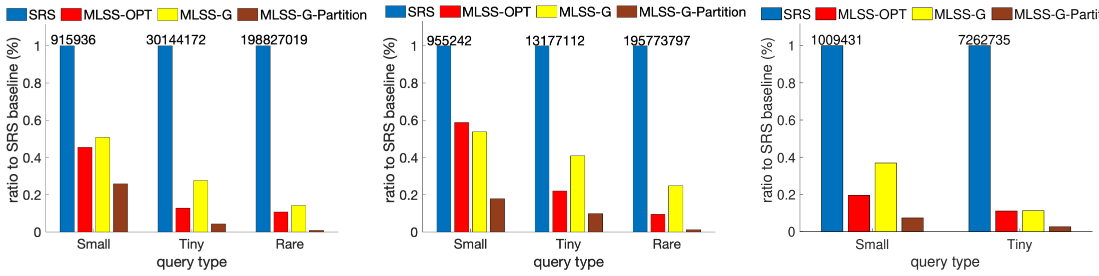
FIGURE 5.10: Trade-off between splitting ratio and MLSS's overall efficiency on Tiny Query.

how sampling parameters of MLSS affect its overall efficiency and how to efficiently fine-tune MLSS in practice. More specifically, there are two parameters to decide when using MLSS – splitting ratio and level partitions. We first study the relationship between splitting ratio and the overall efficiency, and then move to the level partitions. Finally, we experimentally validate our greedy level partition strategy introduced in Section 5.5 and Algorithm 5, and compare it to the theoretical opti-



(a) Queue Model, Small (b) CPP Model, Small (c) Queue Model, Tiny (d) CPP Model, Tiny

FIGURE 5.11: Trade-off between number of levels and MLSS's overall efficiency on Small and Tiny Query.



(a) Queue Model (b) CPP Model (c) RNN Model (single run)

FIGURE 5.12: Efficiency of Greedy Level Partitions.

mal results. By default, we report average numbers over 100 trials, and standard deviations are shown as error bars on top of the plot.

Optimal Splitting Ratios. Figure 5.9 and Figure 5.10 show a clear trade-off between splitting ratio and simulation efforts (to achieve reliable estimate; i.e., 1% CI on Small Query and 10% RE on Tiny Query). To be fair, for splitting ratio from 1 to 7, we all use the "balanced growth" level partition strategy with four levels; that is, the crossing probabilities between consecutive levels are roughly the same. Note that when splitting ratio is 1, MLSS is equivalent to SRS. As shown in Figure 5.9 and Figure 5.10, the first bar (splitting ratio is 1) on both plots matches with the SRS baseline (dashed line). This finding further confirms the relationship between MLSS and SRS as proved in Proposition 15. It is not hard to understand the trade-off. After all, a large splitting ratio directly leads to more splitted sample paths through the simulation process, not to mention that root paths are exponentially copied by the splitting ratio through multiple levels. We can see a clear difference of the optimal splitting ratio between Small Query and Tiny Query, where the latter prefers a larger value since it will potentially create more hits to the target (which is harder to achieve). But interestingly, we can also observe that the optimal choice of splitting ratio (across different models and queries) seems fall in a narrow range around 3. This is the reason why we fix splitting ratio as 3 as the default setting for MLSS.

Optimal Number of Levels. Figure 5.11 shows the relationship between the number of levels and the overall efficiency on Queue Model and CPP Model with Small and Tiny Query, respectively. Here we fix splitting ratio as 3, and for different number of levels, we always adopt the "balanced growth" level partitions. Again, SRS baseline is shown as dashed lines on top of the plots. As these plots suggest, there is also a trade-off between the number of levels and the overall efficiency. Recall the optimal theoretical result about "balanced growth" we introduced in Section 5.5 - Eq(5.26) and Eq(5.27). More levels lead to smaller variance, but more levels also exponentially boost the splitting sample paths in each level of the simulations, which results in the aforementioned trade-off. On the other hand, we observe that there does not seem to exist an universal optimal number of levels across models and queries. For example, Small Query (Figure 5.11(1)-(2)) prefers fewer levels while Tiny Query (Figure 5.11(3)-(4)) requires 5 to 6 levels to achieve optimal performance. This finding is consistent with the observations as we have on optimal splitting ratio from Figure 5.9 and Figure 5.10 that (compared to Small Query) Tiny Query requires more frequent target hits to achieve better performance.

Greedy Level Partitions. After obtaining a better understanding of the factors that affect MLSS's overall efficiency, finally, we provide a practical solution for efficiently fine-tuning MLSS in practice based on our proposed greedy partition strategy introduced in Algorithm 5. Figure 5.12 shows the effectiveness (optimality of level partitions) and efficiency (overall simulation efforts) of the proposed greedy strategy. For better visualizations, we normalize all data as ratios relative to the SRS baseline. Hence in all plots, SRS is shown as the blue bars with ratio 1, and the total number of simulation steps are shown on top of the bar. Red bars (MLSS-OPT) are MLSS with “balanced growth” setting and with optimal number of levels (according to results in Figure 5.11). Yellow bars (MLSS-G) are MLSS with the partition plan returned by our proposed greedy algorithm, and brown bars (MLSS-G-Partition) show the search overhead of greedy partition algorithm. Overall, across all three models and different types of queries, the greedy algorithm is able to automatically search for the near-optimal setting of MLSS – the simulation cost by MLSS-G is not so far away from the optimal (MLSS-OPT), and is still significantly lower than SRS with a 60% to an order-of-magnitude improvement. More importantly, the search overhead (MLSS-G-Partition) of the greedy algorithm is only 10% to 30% compared to the overall simulation cost. Considering that the greedy strategy does not need any information in advance and can automatically search for partition plans, it is a reasonable approach to try in practice if users do not have related knowledge of the model or the query. An interesting by-product of Figure 5.12 is that it also experimentally validates the theory of “balanced growth” setting suggested in [53].

5.6.4 Summary of Experiments

In conclusion, we first demonstrate MLSS’s strong dominance over SRS across different stochastic processes and different types of queries. In general, we observe a query time speedup from 50% (Medium-to-Small queries) up to an order-of-magnitude (Tiny-to-Rare queries), without sacrificing answer quality. Next, we further inspect MLSS and investigate factors that affect its overall efficiency. Finally, we present a greedy strategy that frees users from time-consuming and tedious parameter optimizations. With nearly no information needed in advance, the proposed greedy strategy automatically searches for near-optimal setting (according to the empirically evaluations introduced in Section 5.5.1) simply through simulations. Our experimental results confirm the effectiveness and efficiency of the greedy algorithm — near-optimal query efficiency with only 10% to 30% search overhead of the overall simulation efforts. Nonetheless, the greedy strategy is an alternative remedy if users have no knowledge about the stochastic model or queries. Domain knowledge of the underlying models would definitely be helpful to facilitate the parameter tuning process, but is beyond the scope of this paper.

5.7 Related Work

The closest line of work to ours is query processing over probabilistic databases [31]: range search queries[23, 25, 104, 105], top-k queries [46, 59, 60, 89, 101, 115], join queries [24, 70] and skyline queries [86]. But there is a fundamental difference between these previous studies and our problem. In this paper, we consider query processing on probabilistic temporal data, where temporal dependence is not neglectable when modelling data uncertainty. To meet the new challenge, we propose to model probabilistic temporal data by stochastic processes. As a comparison, previous work on probabilistic databases mainly focuses on the static (snapshot) data, where data uncertainty is considered independently for individuals. Another similar line of work is MCDB and its variants [62, 11, 87, 18]. Unlike probabilistic databases, MCDB does not have strong assumptions about uncertainty independence, but generally embodies data uncertainty with user-defined variable generation (VG) functions. The use of VG functions is analogous to the way that we handle probabilistic temporal data with stochastic processes. Moreover, MCDB's solutions are simulation-based too. The only difference is that our work tweaks sampling procedure to improve sampling efficiency while MCDB focuses on making standard Monte Carlo simulations run faster inside a database management system. In [38], authors used Markov Chains to present uncertain spatio-temporal data and studied how to answer probabilistic range queries. However, their solutions are specific to Markov Chains and requires the transition probability matrix as a priori information. By contrast, our techniques are generally applicable to a variety of stochastic processes, and are largely independent on the underlying model itself.

Regarding durability queries, there are several papers exploring the notions of durability on temporal data. In [73, 74, 107, 41], authors consider durability as a fraction of times (that satisfies certain conditions) over a (temporal) sequence of snapshot data, and answer queries to return the top k objects with highest durability. In [65, 118, 64], authors view durability as the length of time interval. They proposed that, on the two-dimensional space coordinating by durability and data values, skyline queries can discover interesting insights or facts from temporal data that are robust and consistent. Though in different forms, these papers studied durability on existing historical data, which is certain. To the best of our knowledge, our work is among the first to extend the notion of durability into the future, where data can only be probabilistic.

Sampling-based techniques and algorithms play an increasingly important role in the era of big data, ranging from data cleaning [109, 79], integration [75] and evaluation [43], to approximate query processing [21, 62] and visualizations [88]. Some of the work, e.g., [79, 75], share the same idea as ours – going beyond uniform sampling and improving sampling efficiency by properly granting (problem-specific) importance to positive samples. The goal of this line of work is to reduce the total number of samples required without hurting the answer quality. However, in many real-life applications, the actual cost of assessing selected samples (especially that involves manual work, i.e., labeling and annotation) might not be uniform. Thus, the more accurate cost measurement of sampling-based solution in such application scenarios should be the actual cost (time or money) observed from practice, instead of just the number of samples. Based on this consideration, some cost-aware sampling schemes [43] are proposed to practically alleviate the pain of expensive manual work. Another direction is algorithm design for sampling-based solutions. For example, MCDB [62] introduced the concept of tuple-bundle computations to make sampling-based query procedure run faster inside a database management system. In [88], based on online sampling-scheme, authors proposed an optimal incremental visualization algorithm to support rapid and error-free decision making.

Finally, our work also has a deep connection to two classic problems in statistic community: first hitting time [90] and rare event simulation [17]. In statistics, first hitting time (also known as first passage time or survival analysis) is an important feature of stochastic or random process, denoting the amount of time required for a process (starting from an initial state) to reach the threshold for the first time. As mentioned in previous sections, it has a wide applications in very diverse domains [40, 98]. Rare event simulation is the scenario that the probability of the event is low, say, order of 10^{-3} or less. In such cases, the standard Monte Carlo approach would fail to provide reliable estimate in an efficient manner. Importance sampling and splitting-based sampling [45] are two popular variance reduction techniques for rare event simulations. In this paper, we propose to apply Multi-Level Splitting Sampling, which is based on splitting-based sampling, as a query processing technique to efficiently answer durability queries on probabilistic temporal data.

5.8 Conclusion

In this chapter, we have initiated to study the problem of answering durability queries on probabilistic temporal data. We apply stochastic process to model probabilistic temporal data, handling data uncertainty with temporal dependency, and adopt a Monte Carlo approach to answer statistical queries. We propose a novel query processing technique, based on Multi-Level Splitting Sampling, that can efficiently answer durability queries with reliability. We further present an empirical optimization framework that can optimally tune the proposed sampling procedure with little overhead cost. As demonstrated by experiments on a variety of stochastic models and real life applications, our best solution provides up to an order-of-magnitude query time speedup compared to the standard technique, without sacrificing answer quality.

6

Conclusion

Many interesting and practical queries arise from temporal data when incorporating the notion of durability. This dissertation specifically focuses on a set of challenging problems concerning durability queries on temporal data, ranging from two common types of temporal data (sequence-based and instant-stamped) to probabilistic temporal data (generated by a stochastic temporal model). In particular, we provide meaningful and practical interpretations of durability and its corresponding queries on different types of temporal data, and develop efficient and provable techniques for durability query processing.

This dissertation has taken several steps to initiate a systematic study on durability queries on temporal data. Our work not only provides novel insights to the problem from an algorithmic perspective, but also opens up many opportunities towards a unifying framework for durability queries and an end-to-end interactive temporal data analytical system.

As we have shown previously, many durability queries that arise in practice are fundamentally complex, as they often require different combinations of ranking, aggregation, or even solving optimization problems. Previous research [41, 108, 73, 78, 44, 65, 76, 64], including this dissertation, have studied various problems that can be seen as specific instances of durability queries, but there lacks a unifying framework for durability queries that would allow us to abstract above the myriad of ad-hoc solutions and identify reusable algorithmic building blocks for broader classes of durability-related problems. How to design such a general framework for durability queries that unifies common variants of the problem and reusable algorithmic techniques still remains a challenging problem.

On top of the aforementioned framework, there exists the possibility to build an end-to-end interactive temporal data analytical system for the general public. On the one hand, our proposed solutions already achieve interactive-level query response time on large temporal datasets up to the scale of hundreds of millions of records. On the other hand, a unifying and more fundamental understanding of durability queries would further enable a fully automatic pipeline of temporal data analysis – data digesting, query type inference, query processing and result visualization.

To conclude, this dissertation provides a comprehensive and in-depth study into the problem of durability queries on temporal data. We specifically tackle several interesting and challenging durability queries that commonly arise from practical scenarios, and provide efficient query processing techniques with provable worst-case guarantees. While we are still steps away from a unifying framework of durability queries and a fully automatic end-to-end interactive temporal data analytical system, this dissertation has shown considerable promise of a durability-oriented path towards better utilizations of the temporal information behind data, which empowers the general public to harness the increasing value of temporal data.

Appendix A

Appendix

A.1 Full Proofs for Chapter 4

We first introduce some useful notation. Let $dens(t)$ be the density of a timestamp t, i.e., the number of blocking intervals that contain t. Notice that $dens(t)$ is changing as we execute the algorithm. If a point p_i is blocked by at least k records, i.e., $dens(p_i.t) \geqslant k$, at line 7 of Algorithm 4 then we call it an "auxiliary point". Overall, we have that a point can be a solution point, a false check (we run a top-k query but the point does not belong in the solution), or an auxiliary point.

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

Lemma 18. *Let M_i be a set that is empty after the algorithm considering a (auxiliary) point from M_i with density at least k, and let $[l_i, r_i]$ be its corresponding sub-interval. Then one of the two cases hold: The density of each timestamp in $[l_i, r_i]$ is at least k or the algorithm has visited all points in $P([l_i, r_i])$.*

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

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

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

Full proof of Lemma 10.

Proof. Let S^* be the durable points in I. We show that $S \subseteq S^*$ and $S^* \subseteq S$ showing that $S = S^*$. The algorithm always checks by running a top-k query if a point should be in the solution (line 8 of Algorithm 4) so $S \subseteq S^*$.

Next we show the other direction. The algorithm visits the points in descending (on score) order so it is not possible that a point $p \in S^*$ is blocked by at least k records before the algorithm visits p. Before we argue that $S^* \subseteq S$ we also need to make sure that the algorithm does not miss any durable point in a sub-interval $[l_j, r_j]$ that corresponds to an empty set M_j. In Lemma 18 we showed that all timestamps in $[l_j, r_j]$ have density at least k so there is no additional solution point in this sub-interval. Hence $S^* \subseteq S$, and overall we conclude that $S = S^*$. □

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

Lemma 19. *Assume that the algorithm just found the false check p_i. After adding all the blocking intervals from P'_i we have that the density of every timestamp in $[p'_i.t, p_i.t]$ is at least k.*

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

Lemma 20. *Let p_i be a false check and p'_i be the point that it is assigned to. Before adding the k blocking intervals from all points in P'_i (as defined above) we have that either $dens(p'_i.t) \geqslant k$, or $p'_i \in S$ and $dens(p'_i.t) < k$, or $p'_i = a$.*

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

Next, we assume that $p'_i.t \geqslant a$. We prove the lemma by contradiction. Let p'_i be a point that does not belong in S and $dens(p'_i.t) < k$. Notice that $f(p'_i) > f(p_i)$. Since p'_i is not in S it can be either: a false check, an auxiliary point, or a point that the algorithm has not visited before. If p'_i is a false check then from Lemma 19 we have that $dens(p'_i.t) \geqslant k$ at the moment that we found p'_i for first time, which is a contradiction. If p'_i is an auxiliary point then from Lemma 18 we have that $dens(p'_i.t) \geqslant k$, which is a contradiction. If p'_i is a point that the algorithm has not considered before then there are two cases: a) p'_i belongs in an interval $[l_j, r_j]$ of a set M_j that we have removed from M because we have already visited its top-k points. From Lemma 18 we know that $dens(p'_i.t) \geqslant k$, which is a contradiction. b) p'_i belongs in an interval $[l_j, r_j]$ of a set M_j that there still exists in H. Since $f(p'_i) > f(p_i)$ it means that p_i is not the point with the highest score among the sub-intervals that are not removed from M, which is a contradiction.

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

Full proof of Lemma 11.

Proof. If a false check p_i is assigned to a solution point with density less than k then we call it type-1 false check. Otherwise, it is a type-2 false check.

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

Next we focus on type-2 false checks. Let $[l, r]$ be one of the initial disjoint τ-length windows from line 2 of Algorithm 4. We show that after finding k type-2 false checks in $[l, r]$ the density of all timestamps in $[l, r]$ is at least k. If that is the case then the algorithm will not find any other false check in $[l, r]$.

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

Let p_i be a type-2 false check that the algorithm just found in $(t, r]$. Let p'_i be the point that p_i is assigned to, as we defined above. If $p'_i.t \leqslant t$ then we immediately have that $dens(t) \geqslant k$ after adding the at most k new segments from the set P'_i

(Lemma 19), so this case is not interesting. (Notice that if $p'_i.t < a$, before p'_i is set to be a, then this is always the case since $t \geqslant a$).

Now, we assume that for each p_i which is a type-2 false check in $(t, r]$, it holds that $p'_i.t \in (t, p_i.t)$. The main idea to prove that $k_2 \leqslant \chi$ is the following: Each time that the algorithm finds a type-2 false check in $(t, r]$ we find an unmarked interval in X and we mark it. In particular, we show that there always be such an unmarked segment in X with its right endpoint in $[p'_i.t, p_i.t)$. Since p_i is a type-2 false check we have that $dens(p'_i.t) \geqslant k$ and $dens(p_i.t) < k$, at the moment that the algorithm visits p_i (before adding the at most k segments from P'_i). Let Z_1 be the current blocking intervals with right endpoint in $[p'_i.t, p_i.t)$ and $z_1 = |Z_1|$. Let Z_2 be the current blocking intervals with left endpoint in $(p'_i.t, p_i.t]$, and $z_2 = |Z_2|$. Let B be the current blocking intervals with left endpoint in $[p_i.t - \tau, p_i.t]$. We have that $dens(p_i.t) < k \Leftrightarrow |B| < k$, (1). We also have $dens(p'_i.t) \geqslant k \Leftrightarrow |B| - z_2 + z_1 \geqslant k$, (2). From (1), (2), we have that $z_1 > z_2 \Leftrightarrow z_1 \geqslant z_2 + 1$. Notice that the false checks with time instance $\leqslant p'_i.t$ cannot mark a segment in Z_1 because of the description of the marking process. Furthermore, a previous false check with timestamp at the right of $p_i.t$ cannot mark a segment in Z_1: Let p_j be a false check that the algorithm found in a previous iteration in $(p_i.t, r]$ and let p'_j be the point that it is assigned to. If $p'_j.t > p_i.t$ then the marking process does not mark any segment in Z_1. Otherwise, if $p'_j.t \leqslant p_i.t$ then the density of all points in $[p'_j.t, p_i.t] \cup [p_i.t, p'_j.t]$ would be at least k after the algorithm adds the segments from P'_j, which is a contradiction because $dens(p_i.t) < k$ when we visit p_i. Hence only false checks in $(p'_i.t, p_i.t]$ can mark segments in Z_1. Recall that Z_2 are the current segments with left endpoints in $(p'_i.t, p_i.t]$. Even if all segments in Z_2 were created by type-2 false checks and even if all of them mark segments from Z_1, we showed that $z_1 \geqslant z_2 + 1$, so we can always find a new unmarked segment in Z_1. Notice that any segment in Z_1 has its right endpoint in $[p'_i.t, p_i.t)$ and since all the segments have length τ, they contain t and hence they belong in X. Each time that we find a type-2 false check in $(t, r]$ we mark a new segment in X, so $k_2 \leqslant \chi$ and we conclude that $dens(t) \geqslant k$.

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

Full proof of Lemma 13.

Proof. We show the result extending the main ideas from [13]. Let $P(I) = \{p_{j+1}, \ldots, p_{j+L}\}$. For $p_i \in P(I)$, let X_i be a random variable which is 1 if $p_i \in \mathcal{C}$, and 0 otherwise. From linearity of expectation we have that $\mathbb{E}[|C|] = \mathbb{E}\left[\sum_{i=j+1}^{j+|I|} X_i\right] = \sum_{i=j+1}^{j+|I|} \mathbb{E}[X_i] = \sum_{i=j+1}^{j+|I|} \mathbf{Pr}[X_i = 1]$. We focus on computing $\mathbf{Pr}[X_i = 1]$. Let $P_i = P([p_{i-\tau}.t, p_i.t]) = \{p_{i-\tau}, \ldots, p_{i-1}, p_i\}$. By independence we have that the probability of each point in P_i to be in the k-skyband of P_i is the same, so we can compute $\mathbf{Pr}[X_i = 1]$ by first finding the expected size of the k-skyband in P_i and then divide it by the number of points, $\tau + 1$.

Let B_i be the k-skyband of the $\tau + 1$ points P_i. Let $V_j \subset N$ for $1 \leqslant j \leqslant d$, with $|V_j| = \tau + 1$ such that V_j contains the values that are assigned to the j-th coordinate of the points in P_i. We compute $\mathbb{E}[|B_i| \mid V_1, \ldots, V_d]$. Let $A(\tau + 1, d)$ be the expected size of the k-skyband of a set $\bar{P}$ with $\tau + 1$ points in $\mathbb{R}^d$ in the d-dimensional random permutation model. Notice that $A(\tau + 1, d) = \mathbb{E}[|B_i| \mid V_1, \ldots, V_d]$. We compute $A(\tau + 1, d)$ as follows. From linearity of expectation we can compute the probability that a point in $\bar{P}$ belongs in the k-skyband and take the sum of them, $A(\tau + 1, d) = \sum_{\bar{p} \in \bar{P}} \mathbf{Pr}\left[\bar{p} \in k\text{-skyband of } \bar{P}\right]$. Assume that a point $\bar{p} \in \bar{P}$ has the g-th largest first coordinate among the points in $\bar{P}$. Notice that this can happen with probability $\frac{1}{\tau+1}$. Since the first coordinate of the g-th point ($\bar{p}$) is greater than the first coordinates of $g-1$ points it cannot be dominated by any of those. Therefore, the g-th point belongs in the k-skyband if and only if its remaining $d-1$ coordinates belong in the k-skyband among the points in $\bar{P}$ with the g-th through the $(\tau+1)$-th largest first coordinate. The probability that the g-th point is in the k-skyband is, by independence, the expected number of the k-skyband in the remaining points and coordinates, which is $A(\tau+1-g+1, d-1)$, divided by the total number of the remaining points in the set which are $\tau+1-g+1$. Notice that $A(k', y) = k'$ for $k' \leqslant k$ and any y. Hence, we have $A(\tau+1, d) = \sum_{j=1}^{\tau+1} \sum_{g=1}^{\tau+1} \frac{1}{\tau+1} \frac{A(\tau+1-g+1,d-1)}{\tau+1-g+1} = \frac{1}{\tau+1} \sum_{j=1}^{\tau+1} \sum_{J=1}^{\tau+1} \frac{A(J,d-1)}{J} = \sum_{J=1}^{\tau+1} \frac{A(J,d-1)}{J}$. Notice that $A(x, y)$ is monotonically increasing in x, so if $x_1 \leqslant x_2$, then $A(x_1, y) \leqslant A(x_2, y)$. Furthermore, we note that $A(\tau+1, 1) = k$ since in one dimension the top-k points belong in the k-skyband. We have, $A(\tau+1, d) = \sum_{J=1}^{\tau+1} \frac{A(J,d-1)}{J} \leqslant A(\tau+1, d-1) \sum_{J=1}^{\tau+1} \frac{1}{J} \leqslant A(\tau+1, d-1) O(\log \tau)$. Iterating this recurrence on d until $A(\tau+1, 1) = k$ gives the upper bound $A(\tau+1, d) = O(k \log^{d-1} \tau)$.

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

Bibliography

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

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] 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.

[4] P. K. Agarwal, L. Arge, J. Erickson, P. G. Franciosa, and J. S. Vitter. Efficient searching with linear constraints. *Journal of Computer and System Sciences*, 61(2):194–216, 2000.

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

[6] P. K. Agarwal, S. Har-Peled, H. Kaplan, and M. Sharir. Union of random minkowski sums and network vulnerability analysis. *Discrete & Computational Geometry*, 52(3):551–582, 2014.

[7] P. K. Agarwal, H. Kaplan, and M. Sharir. Union of hypercubes and 3d minkowski sums with random sizes. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[8] P. K. Agarwal, N. Kumar, S. Sintos, and S. Suri. Range-max queries on uncertain data. *Journal of Computer and System Sciences*, 94:118–134, 2018.

[9] P. K. Agarwal and J. Matoušek. Dynamic half-space range reporting and its applications. *Algorithmica*, 13(4):325–345, 1995.

[10] H. Albrecher, J.-F. Renaud, and X. Zhou. A lévy insurance risk process with tax. *Journal of Applied Probability*, 45(2):363–375, 2008.

[11] S. Arumugam, F. Xu, R. Jampani, C. Jermaine, L. L. Perez, and P. J. Haas. Mcdb-r: Risk analysis in the database. *Proceedings of the VLDB Endowment*, 3(1-2):782–793, 2010.

[12] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155, 2003.

[13] J. L. Bentley, H.-T. Kung, M. Schkolnick, and C. D. Thompson. On the average number of maxima in a set of vectors and applications. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1977.

[14] K. Binder, D. Heermann, L. Roelofs, A. J. Mallinckrodt, and S. McKay. Monte carlo simulation in statistical physics. *Computers in Physics*, 7(2):156–157, 1993.

[15] C. M. Bishop. Mixture density networks. 1994.

[16] 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.

[17] J. Bucklew. *Introduction to rare event simulation*. Springer Science & Business Media, 2013.

[18] Z. Cai, Z. Vagena, L. Perez, S. Arumugam, P. J. Haas, and C. Jermaine. Simulation of database-valued markov chains using simsql. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 637–648, New York, NY, USA, 2013. Association for Computing Machinery.

[19] T. M. Chan. Three problems about dynamic convex hulls. *International Journal of Computational Geometry & Applications*, 22(04):341–364, 2012.

[20] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *ACM Sigmod Record*, volume 29, pages 391–402. ACM, 2000.

[21] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519, 2017.

[22] B. Chazelle, L. J. Guibas, and D.-T. Lee. The power of geometric duality. *BIT Numerical Mathematics*, 25(1):76–90, 1985.

[23] R. Cheng, D. V. Kalashnikov, and S. Prabhakar. Evaluating probabilistic queries over imprecise data. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 551–562, 2003.

[24] R. Cheng, S. Singh, S. Prabhakar, R. Shah, J. S. Vitter, and Y. Xia. Efficient join processing over uncertain data. In *Proceedings of the 15th ACM international conference on Information and knowledge management*, pages 738–747, 2006.

[25] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, and J. S. Vitter. Efficient indexing methods for probabilistic threshold queries over uncertain data. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 876–887, 2004.

[26] F. Chollet et al. keras, 2015.

[27] W. G. Cochran. *Sampling techniques*. John Wiley & Sons, 2007.

[28] R. B. Cooper. Queueing theory. In *Proceedings of the ACM'81 conference*, pages 119–122, 1981.

[29] 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.

[30] D. R. Cox and H. D. Miller. *The theory of stochastic processes*, volume 134. CRC press, 1977.

[31] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Communications of the ACM*, 52(7):86–94, 2009.

[32] G. Das, D. Gunopulos, N. Koudas, and N. Sarkas. Ad-hoc top-k query answering for data streams. In *Proceedings of the 33rd international conference on Very large data bases*, pages 183–194. Citeseer, 2007.

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

[34] M. De Berg, M. Van Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[35] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein. A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67, 2005.

[36] D. Eck and J. Schmidhuber. A first look at music composition using lstm recurrent neural networks. Technical report, 2002.

[37] H. Edelsbrunner and J. Harer. *Computational topology: an introduction.* American Mathematical Soc., 2010.

[38] T. Emrich, H.-P. Kriegel, N. Mamoulis, M. Renz, and A. Zufle. Querying uncertain spatio-temporal data. In *2012 IEEE 28th international conference on data engineering*, pages 354–365. IEEE, 2012.

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

[40] P. Fauchald and T. Tveraa. Using first-passage time in the analysis of area-restricted search and habitat selection. *Ecology*, 84(2):282–288, 2003.

[41] J. Gao, P. K. Agarwal, and J. Yang. Durable top-k queries on temporal data. *Proceedings of the VLDB Endowment*, 11(13):2223–2235, 2018.

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

[43] J. Gao, X. Li, Y. E. Xu, B. Sisman, X. L. Dong, and J. Yang. Efficient knowledge graph accuracy evaluation. *Proceedings of the VLDB Endowment*, 12(11):1679–1691, 2019.

[44] J. Gao, S. Sintos, P. K.Agarwal, and J. Yang. Durable top-k instant-stamped temporal records with user-specified scoring functions. Technical report, Duke University, 2020. `https://users.cs.duke.edu/~jygao/VLDB2020_full.pdf`.

[45] M. J. J. Garvels. The splitting method in rare event simulation. 2000.

[46] T. Ge, S. Zdonik, and S. Madden. Top-k queries on uncertain data: on score distribution and typical answers. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 375–388, 2009.

[47] P. W. Glynn and D. L. Iglehart. Importance sampling for stochastic simulations. *Management science*, 35(11):1367–1392, 1989.

[48] G. Goel and A. Mehta. Online budgeted matching in random input models with applications to adwords. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 982–991. Society for Industrial and Applied Mathematics, 2008.

[49] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[50] A. Graves and J. Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

[51] G. Grimmett, G. R. Grimmett, D. Stirzaker, et al. *Probability and random processes*. Oxford university press, 2001.

[52] S. Har-Peled and B. Raichel. On the complexity of randomly weighted multiplicative voronoi diagrams. *Discrete & Computational Geometry*, 53(3):547–568, 2015.

[53] T. E. Harris. The theory of branching process. 1964.

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

[55] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[56] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[57] 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.

[58] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *The VLDB Journal*, 13(1):49–70, 2004.

[59] M. Hua, J. Pei, and X. Lin. Ranking queries on uncertain data. *The VLDB Journal*, 20(1):129–153, 2011.

[60] M. Hua, J. Pei, W. Zhang, and X. Lin. Efficiently answering probabilistic threshold top-k queries on uncertain data. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1403–1405. IEEE, 2008.

[61] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys (CSUR)*, 40(4):11, 2008.

[62] R. Jampani, F. Xu, M. Wu, L. L. Perez, C. Jermaine, and P. J. Haas. Mcdb: a monte carlo approach to managing uncertain data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 687–700, 2008.

[63] J. Jestes, J. M. Phillips, F. Li, and M. Tang. Ranking large temporal data. *Proceedings of the VLDB Endowment*, 5(11):1412–1423, 2012.

[64] B. Jiang and J. Pei. Online interval skyline queries on time series. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1036–1047. IEEE, 2009.

[65] X. Jiang, C. Li, P. Luo, M. Wang, and Y. Yu. Prominent streak discovery in sequence data. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1280–1288. ACM, 2011.

[66] C. Jin, K. Yi, L. Chen, J. X. Yu, and X. Lin. Sliding-window top-k queries on uncertain streams. *Proceedings of the VLDB Endowment*, 1(1):301–312, 2008.

[67] M. I. Jordan. Serial order: A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

[68] H. Kahn and T. E. Harris. Estimation of particle transmission by random sampling. *National Bureau of Standards applied mathematics series*, 12:27–30, 1951.

[69] J. G. Kemeny and J. L. Snell. *Markov chains.* Springer-Verlag, New York, 1976.

[70] B. Kimelfeld and Y. Sagiv. Maximally joining probabilistic data. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 303–312, 2007.

[71] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models.* Prentice-Hall, Inc., 1984.

[72] P. L'Ecuyer, V. Demers, and B. Tuffin. Splitting for rare-event simulation. In *Proceedings of the 2006 winter simulation conference*, pages 137–148. IEEE, 2006.

[73] 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.

[74] U. Leong Hou, N. Mamoulis, K. Berberich, and S. Bedathur. Durable top-k search in document archives. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

[75] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*, pages 615–629, 2016.

[76] F. Li, K. Yi, and W. Le. Top-k queries on temporal data. *The VLDB Journal—The International Journal on Very Large Data Bases*, 19(5):715–733, 2010.

[77] M. Mahdian and Q. Yan. Online bipartite matching with random arrivals: an approach based on strongly factor-revealing lps. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 597–606. ACM, 2011.

[78] 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.

[79] N. G. Marchant and B. I. Rubinstein. In search of an entity resolution oasis: optimal asymptotic sequential importance sampling. *PVLDB*, 10(11):1322–1333, 2017.

[80] J. Matousek. Reporting points in halfspaces. *Computational Geometry*, 2(3):169–186, 1992.

[81] A. Mehta, A. Saberi, U. Vazirani, and V. Vazirani. Adwords and generalized online matching. In *46th Annual IEEE Symposium on Foundations of Computer Science (FOCS'05)*, pages 264–273. IEEE, 2005.

[82] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous monitoring of top-k queries over sliding windows. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 635–646. ACM, 2006.

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

[84] C. Newell. *Applications of queueing theory*, volume 4. Springer Science & Business Media, 2013.

[85] M. O'Kelly, A. Sinha, H. Namkoong, R. Tedrake, and J. C. Duchi. Scalable end-to-end autonomous vehicle testing via rare-event simulation. In *Advances in Neural Information Processing Systems*, pages 9827–9838, 2018.

[86] J. Pei, B. Jiang, X. Lin, and Y. Yuan. Probabilistic skylines on uncertain data. In *Proceedings of the 33rd international conference on Very large data bases*, pages 15–26. Citeseer, 2007.

[87] L. L. Perez, S. Arumugam, and C. M. Jermaine. Evaluation of probabilistic threshold queries in mcdb. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 687–698, 2010.

[88] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfield. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, Aug. 2017.

[89] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 886–895. IEEE, 2007.

[90] S. Redner. *A guide to first-passage processes.* Cambridge University Press, 2001.

[91] R. Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[92] R. Y. Rubinstein and D. P. Kroese. *Simulation and the Monte Carlo method*, volume 10. John Wiley & Sons, 2016.

[93] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[94] B. Salzberg and V. Tsotras. A comparision of access methods for temporal data. Technical report, Technical Report TR-18, Time Center: Aalborg University–Denmark and . . . , 1997.

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

[96] K. Semertzidis and E. Pitoura. Durable graph pattern queries on historical graphs. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 541–552. IEEE, 2016.

[97] K. Semertzidis and E. Pitoura. Top-k durable graph pattern queries on temporal graphs. *IEEE Transactions on Knowledge and Data Engineering*, 31(1):181–194, 2018.

[98] A. N. Shiryaev. *Essentials of stochastic finance: facts, models, theory*, volume 3. World scientific, 1999.

[99] R. H. Shumway and D. S. Stoffer. *Time series analysis and its applications: with R examples.* Springer, 2017.

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

[101] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang. Top-k query processing in uncertain databases. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 896–905. IEEE, 2007.

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

[103] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass. *Temporal databases: theory, design, and implementation*. Benjamin-Cummings Publishing Co., Inc., 1993.

[104] Y. Tao, R. Cheng, X. Xiao, W. K. Ngai, B. Kao, and S. Prabhakar. Indexing multi-dimensional uncertain data with arbitrary probability density functions. In *VLDB*, volume 5, pages 922–933. Citeseer, 2005.

[105] Y. Tao, X. Xiao, and R. Cheng. Range search on multidimensional uncertain data. *ACM Transactions on Database Systems (TODS)*, 32(3):15–es, 2007.

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

[107] 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, 2013.

[108] 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.

[109] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 469–480, New York, NY, USA, 2014. Association for Computing Machinery.

[110] G. Whitmore. First-passage-time models for duration data: regression structures and competing risks. *Journal of the Royal Statistical Society: Series D (The Statistician)*, 35(2):207–219, 1986.

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

[112] Y. Xu. First exit times of compound poisson processes with parallel boundaries. *Sequential Analysis*, 31(2):135–144, 2012.

[113] J. Yang. *Temporal data warehousing.* 2001.

[114] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *Proceedings 17th International Conference on Data Engineering*, pages 51–60. IEEE, 2001.

[115] K. Yi, F. Li, G. Kollios, and D. Srivastava. Efficient processing of top-k queries in uncertain databases. In *2008 IEEE 24th International Conference on Data Engineering*, pages 1406–1408. IEEE, 2008.

[116] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 189–200. IEEE, 2003.

[117] D. Zhang, A. Markowetz, V. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 237–245, 2001.

[118] G. Zhang, X. Jiang, P. Luo, M. Wang, and C. Li. Discovering general prominent streaks in sequence data. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(2):9, 2014.

Biography

Junyang Gao was born in 1993, China.

He earned a Bachelor's degree of Engineering in Computer Science & Technology in 2015 from the Department of Computer Science at Tsinghua University, and a Master's of Science from the Department of Computer Science at Duke University in 2018. He completed the Doctor of Philosophy in Computer Science, advised by Professor Jun Yang, also from the Department of Compuater Science at Duke University in September, 2020.

Junyang Gao will join Google Inc. in the New York City office after graduation.