Ranking Queries over Range Data

A Thesis

submitted to the designated
by the General Assembly
of the Department of Computer Science and Engineering
Examination Committee

by

Georgios Kotsinas

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

IN DATA AND COMPUTER SYSTEMS ENGINEERING

WITH SPECIALIZATION

IN DATA SCIENCE AND ENGINEERING

University of Ioannina
June 2024

Examining Committee:

- **Nikolaos Mamoulis**, Professor, Computer Science and Engineering Department, University of Ioannina (Supervisor).
- **Panagiotis Vasiliadis**, Professor, Computer Science and Engineering Department, University of Ioannina.
- **Apostolos Zarras**, Professor, Computer Science and Engineering Department, University of Ioannina.

Dedication

I would like to dedicate this thesis to my father.

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Professor Nikolaos Mamoulis, for his invaluable guidance, support, and encouragement throughout the course of my research. His knowledge and insightful feedback have been instrumental in shaping this thesis. I am also profoundly thankful to my mentors, George Christodoulou and Panagiotis Bouros, whose expertise and advice have been crucial to the progression of my work. Their willingness to share their knowledge and their constant support have been a source of inspiration and motivation. Lastly, I wish to extend my heartfelt thanks to my family. Their unwavering support, patience, and understanding have been my foundation throughout this journey. Their love and belief in my abilities have given me the strength to persevere and achieve my goal.

Ioannina, June 2024
Georgios Kotsinas

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Georgios Kotsinas, M.Sc. in Computer Science, Department of Computer Science and Engineering. University of Ioannina, Greece, June 2024.
Ranking Queries over Range Data.
Advisor: Nikolaos Mamoulis, Professor.

Today's data-driven world has made it essential to manage and analyze large volumes of temporal data efficiently. This thesis addresses the problem of identifying the top k time intervals that best intersect with a query interval within a given temporal data domain. In pursuit of addressing this issue with maximal efficiency, we further develop the HINTm index to support ranking queries.
HINTm, is a Hierarchical Index for Intervals in arbitrary domains designed for main memory and defines a hierarchical domain decomposition which assigns each interval to at most two partitions per level. It has previously been recognized as the most efficient interval index in the literature, has undergone numerous optimizations to avoid unnecessary comparisons and expedite range query responses over extensive collections of intervals. Building on its optimizations, this work adapted HINTm to effectively handle top k queries.
The ranking criterion is defined by the absolute interval intersection, enabling the identification of intervals that intersect better with a given query interval. Except from the naive approach that simply traverses the index and scans its partitions for results, various methods were developed to prioritize partitions that contain larger intervals first. In reference, "Top-down", "Depth-first", "Ordered" and "Sorted" traversals aim to optimize the processing speed of top k queries. Additionally, a pruning mechanism was implemented to bypass scanning index partitions that are guaranteed not to contain intervals of the final set. This pruning mechanism, termed "Upper bounds", was deployed in two distinct versions. The first version assigns a static Upper bound to each index partition based on the partition's endpoints. The second, an updated version, incorporates the metadata information of the maximum interval within each partition.

Extensive experiments were conducted on four datasets with varying characteristics, measuring the number of queries executed per second. These experiments aimed to understand system scalability concerning different query extents and values of k. The results indicate that larger query extents and higher values of k are associated with reduced throughput. However, the application of the “Upper bounds” accelerates the overall process. Finally, metadata Upper bounds provide even better performance, always with respect to the diverse characteristics of datasets being utilized.

Εκτεταμενη περιληψη

Γεώργιος Κοτσίνας, Δ.Μ.Σ. στη Μηχανική Δεδομένων και Υπολογιστικών Συστημάτων, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, Ιούνιος 2024.
Ερωτήματα Κατάταξης σε Δεδομένα Εύρους.
Επιβλέπων: Νικόλαος Μαμουλής, Καθηγητής.

Στην παρούσα εποχή, όπου τα δεδομένα κυριαρχούν, η αποδοτική διαχείριση και ανάλυση μεγάλων όγκων χρονικών δεδομένων αποτελεί ζήτημα υψίστης σημασίας. Αυτή η μεταπτυχιακή διπλωματική εργασία πραγματεύεται το πρόβλημα της αναγνώρισης των κορυφαίων *k* χρονικών διαστημάτων που επικαλύπτουν καλύτερα ένα δοθέν διάστημα-ερώτημα εντός ενός δεδομένου χρονικού τομέα. Στην προσπάθεια να αντιμετωπιστεί αυτό το ζήτημα με μέγιστη αποδοτικότητα, αναπτύξαμε περαιτέρω το ευρετήριο HINTm για να υποστηρίξει ερωτήματα κατάταξης.
Το HINTm, είναι ένα ιεραρχικό ευρετήριο για διαστήματα σε ακαθόριστους τομείς. Σχεδιασμένο για την κύρια μνήμη ορίζει μια ιεραρχική διχοτόμηση του τομέα, η οποία αναθέτει κάθε διάστημα σε το πολύ δύο διαμερίσματα ανά επίπεδο. Έχει αναγνωριστεί ως το πιο αποδοτικό ευρετήριο διαστημάτων στη βιβλιογραφία και έχει υποστεί πολυάριθμες βελτιστοποιήσεις για την αποφυγή περιττών συγκρίσεων. Διακρίνεται για την ταχύτητα των απαντήσεων ερωτημάτων εύρους σε μεγάλες συλλογές από διαστήματα. Βασιζόμενοι στις βελτιστοποιήσεις του, προσαρμόσαμε τον HINTm για να διαχειρίζεται αποτελεσματικά ερωτήματα κατάταξης (top k).
Το κριτήριο κατάταξης ορίζεται ως η επικάλυψη διαστήματος ευρετηρίου με διάστημα ερωτήματος, επιτρέποντας την αναγνώριση των *k* διαστημάτων που επικαλύπτουν καλύτερα το δοθέν διάστημα του ερωτήματος. Εκτός από την απλή μέθοδο που διασχίζει το ευρετήριο και σαρώνει τα διαμερίσματά του για αποτελέσματα, αναπτύχθηκαν διάφορες μέθοδοι διάσχισης ευρετηρίου για να δώσουν προτεραιότητα σε διαμερίσματα του που περιέχουν μεγαλύτερα διαστήματα. Αναφορικά, οι μέθοδοι «Από πάνω προς τα κάτω», «Πρώτα σε βάθος», «Ταξινομημένη» και «Διατεταγμένη» διάσχιση έχουν ως στόχο να βελτιώσουν την ταχύτητα επεξεργασίας των ερωτημάτων κατάταξης. Επιπλέον, υλοποιήθηκε ένας μηχανισμός

κλαδέματος για να παρακάμπτει τη σάρωση διαμερισμάτων του ευρετηρίου που είναι εγγυημένο ότι δεν περιέχουν διαστήματα του τελικού συνόλου. Αυτός ο μηχανισμός κλαδέματος, ονομαζόμενος «Ανώτατα όρια», αναπτύχθηκε σε δύο διακριτές εκδοχές. Η πρώτη εκδοχή αναθέτει ένα στατικό ανώτατο όριο σε κάθε διαμέρισμα του ευρετηρίου με βάση τα άκρα του διαμερίσματος. Η δεύτερη πιο βελτιωμένη εκδοχή, ενσωματώνει την μεταπληροφορία του μέγιστου διαστήματος μέσα σε κάθε διαμέρισμα υπολογίζοντας έτσι το ανώτατο όριο κάθε διαμερίσματος. Διεξήχθησαν εκτενείς πειραματισμοί σε τέσσερα σύνολα δεδομένων με διαφορετικά χαρακτηριστικά, μετρώντας τον αριθμό των εκτελούμενων ερωτημάτων ανά δευτερόλεπτο. Τα πειράματα αποσκοπούσαν στην διερεύνηση της κλιμάκωσης του συστήματος σε σχέση με διαφορετικές εκτάσεις ερωτημάτων και τιμές του k. Τα αποτελέσματα υποδεικνύουν ότι οι μεγαλύτερες εκτάσεις ερωτημάτων και οι υψηλότερες τιμές του k συνδέονται με μειωμένη απόδοση. Ωστόσο, η εφαρμογή των «Ανώτατων ορίων» επιταχύνει τη συνολική διαδικασία. Τέλος, τα ανώτατα όρια μεταδεδομένων παρέχουν ακόμα καλύτερη απόδοση, πάντα σε σχέση με τα ποικίλα χαρακτηριστικά των συνόλων δεδομένων που χρησιμοποιήθηκαν.

CHAPTER 1

INTRODUCTION

Effective temporal data management solutions are critically needed to address the complexities and demands of handling time-variant information in modern data-driven applications. Temporal data, characterized by its time-varying nature, is ubiquitous across various domains such as transport, healthcare, finance, social media and sensor networks. Additionally, several applications call for the management of big interval collections. Each tuple in a temporal database has a validity interval that represents the amount of time the tuple is valid (duration) [1]. In data anonymization, attribute values are frequently generalized into value ranges [2]. Interval search is a module used in many computational geometry problems [3] (e.g., windowing). Intervals can be used to model and control the internal states of window queries in stream processors [4].

In the realm of temporal data analysis, the challenge of efficiently ranking intervals within a dataset is a critical problem with wide-ranging applications. From temporal data analysis in financial markets to genomic range queries in bioinformatics, the need to quickly and accurately rank intervals from vast collections is increasingly essential. This thesis addresses this challenge through the development of novel techniques for ranking queries over range data, specifically focusing on identifying the top k intervals based on a specified ranking criterion.

The core problem can be succinctly stated: given a collection of intervals and a query interval, how can we efficiently determine the top k intervals that have the most significant intersection with the query? This problem is pivotal in scenarios where rapid response times and high precision are crucial, such as real-time monitoring systems, database querying, and various temporal data management applications. Some instances of these queries across various data domains are as follows:

- Healthcare, find the k larger periods of symptoms that a patient had during an episode of medical concern.
- Weather monitoring, find the top k time intervals of significant weather events that best intersect with a given period of abnormal weather conditions.
- Traffic analysis, find the top k intervals of heavy traffic congestion that best intersect with a given period of a traffic incident.
- Social media analysis, find the top k periods of high social media activity that best intersect with a given event or campaign duration.
- Energy consumption, find the top k intervals of high energy usage that best intersect with a given period of peak demand.
- Network security, find the top k intervals of high network activity that best intersect with a given period of a security breach.
- Market analysis, find the top k periods of high sales activity that best intersect with a given promotional event duration.

Despite the importance and broad application of ranking queries over temporal data, several challenges persist. Firstly, the sheer volume of data necessitates efficient indexing and querying mechanisms to ensure timely responses. Secondly, the temporal aspect introduces the need for time-aware methods that can efficiently manage and query temporal datasets while maintaining accuracy. This thesis aims to contribute to this field by investigating novel methods for efficiently processing top k queries over temporal data using the preexisting work of HINTm [5] a hierarchical index for intervals that can handle valid time data, suitable for applications that manage large collections of intervals.

HINTm indexes a large collection S of objects (or records) based on an interval attribute that characterizes each object. Each object $s \in S$ is modeled as a triple ($s.id$, $s.st$, $s.end$), where $s.id$ is the object's identifier, which can be used to access any other attribute of the object, and $[s.st,\ s.end]$ represents the interval's endpoints. HINTm also uses various optimizations to accelerate its performance. To date, HINTm has supported range queries and Allen's interval relations [6]. The present

research concentrates on the appliance of ranking queries in the domain of HINTm, while we endeavor to leverage its optimizations.
Given a query interval $q = [q.st. q.end]$, and a positive integer k, the objective is to find the top k objects that belong to S and overlap with q by using the index of HINTm. The ranking of them will be determined by their absolute overlapping duration score. This score can be easily computed for every interval s_i that intersects with the query q, by the following, $\min(q.end,\ s_i.end) - \max(q.st,\ s_i.st)$. The following figure 1.1 illustrates the problem's formation in a straightforward manner. The task involves selecting the three intervals that have the highest degree of overlap with a given query within a specified domain of intervals.

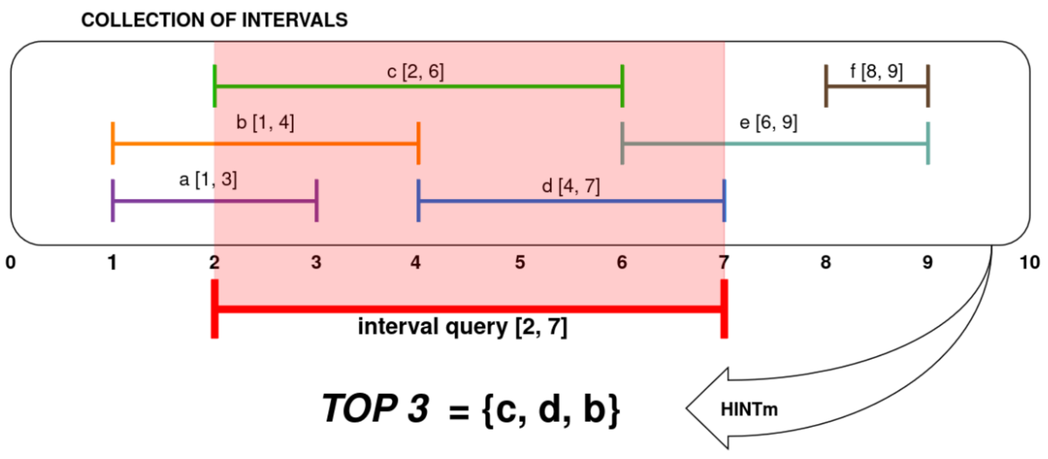


Figure 1.1: Example of top 3

We investigate various methodologies for traversing the optimized index of HINTm aiming to reduce the computational load while searching for the top k records. This exploration is undertaken with consideration of the diverse characteristics of the index as well as of the datasets being utilized. Finally, we conduct experiments with various query extents across different values of k. For this purpose, we utilized four datasets with distinct characteristics.

1.1 Contributions

In summary, this thesis makes the following contributions:

- We further developed the Hierarchical Index for Intervals in arbitrary domains (HINTm) to support top k queries.
- We proposed traversal methods for the index that prioritize larger intervals first, along with two versions of a pruning mechanism that accelerate the execution of top k queries.
- We evaluated our methods using four real datasets with distinct characteristics, conducting experiments on different query ranges and values of k.

1.2 Outline

The rest of this paper is organized as follows: Chapter 2 reviews the existing literature on top k processing and interval indexing, providing a comprehensive overview of related methods and techniques with a particular focus on the HINTm index and its optimizations. Chapter 3 formally defines the problem addressed in this thesis and the criteria used for ranking the intervals based on their absolute overlap duration with the query interval. Chapter 4 details the methods for traversing the index, incorporating the pruning mechanism of "Upper bounds" to enhance the efficiency of the query process. Chapter 5 presents an extensive experimental analysis conducted on various datasets, exploring the performance of the proposed methods with different query extents and various values of k. Chapter 6 summarizes the key findings of the thesis and discusses potential directions for future research.

CHAPTER 2

RELATED WORKS

2.1 Top-k Processing

Top-k processing is a crucial aspect in databases and information retrieval systems, where the objective is to efficiently retrieve the top k objects with the highest overall scores from ranked inputs. Several approaches and algorithms have been proposed to address this challenge, focusing on reducing computational cost, memory usage, and the number of object accesses.

A seminal work in this domain is by Fagin et al. [7], which introduces the concept of aggregating scores from multiple attributes to determine the top k objects. Each object in the database has multiple scores, one for each attribute, and these scores are combined using a monotone aggregation function such as min or average. The naive approach to this problem requires accessing every object in the database to compute its combined score, which is inefficient. Fagin's Algorithm (FA) provides a more efficient solution for certain monotone aggregation functions. However, FA has

limitations, including the requirement for large, potentially unbounded buffers as the database size increases. To address these limitations, Fagin et al. propose the Threshold Algorithm (TA). TA is optimal for all monotone aggregation functions and operates efficiently across all databases. Unlike FA, TA maintains a small, constant-size buffer, making it more scalable. Additionally, TA supports early stopping, allowing for an approximate version of the top k results when exact precision is not necessary.
Mamoulis et al. [8] build upon these foundations and propose enhancements to further optimize top-k queries. Their work identifies two critical phases that any top-k algorithm based on sorted accesses must undergo. Leveraging these phases, they introduce a new algorithm designed to minimize the number of object accesses, computational cost, and memory requirements for top-k searches using monotone aggregate functions. A key contribution of their research is the development of a multiway top-k join operator, which offers significant advantages over traditional evaluation trees constructed from binary top-k join operators. This operator improves the efficiency of combining multiple ranked inputs by reducing redundancy and the number of intermediate computations. Moreover, they explore the concept of top-k cubes and their efficient computation, which facilitates the implementation of roll-up operations in multi-dimensional top-k queries. The proposed methods demonstrate superior performance compared to previous techniques. Their approach accesses fewer objects and achieves faster execution times, highlighting the practical benefits of their optimizations in real-world applications.

2.2 Interval Indexing

This subsection discusses epigrammatically the main-memory indices for intervals employed by the authors of HINT/HINTm for comparative analysis of their work. Interval indexing has seen various innovative approaches aimed at efficiently managing and querying interval data. Among these, the interval tree developed by Edelsbrunner [9] stands out as a widely utilized data structure. This tree is suited for stabbing and range queries, organizing intervals around a center point to balance the tree. Intervals that include the center point are stored at the root, while left and right subtrees handle intervals before and after the center, respectively. Two lists sorted by interval start and end values are maintained at each node. The interval tree requires numerous comparisons for most range query results, which is a notable drawback.

A simpler yet effective structure is the 1D-grid, which divides the data domain into non-overlapping partitions. Each interval is assigned to all partitions it overlaps with, ensuring comprehensive coverage of the data domain. However, this approach can lead to duplicate results if a query intersects multiple partitions, complicating the retrieval process [10].

Another structure is the period index [11], which considers both interval durations and values. This self-adaptive structure, like a 1D-grid, partitions the time domain and hierarchically organizes intervals within each partition based on their locations and durations. This method effectively supports range and duration queries.

The timeline index [12], derived from the time index [13], is designed for general-purpose temporal data access. It maintains a sorted event list (table of triples [$time,$ $id, isStart$] denoting the interval's timing, identifier, and whether it is a start or end point) of all interval endpoints, prioritized by time and secondly by $isStart$ (11 if the triple refers to start else 0 for an end) in descending order. Checkpoints define specific timestamps where all intervals that overlap with it are considered a whole. To process a range query, the method locates the nearest checkpoint preceding the query start and initializes an active set of its intervals. The event list is then scanned from this checkpoint, updating the set with intervals starting or ending at this event. This method, however, can be inefficient as it often accesses more data and makes more comparisons than necessary.

Among these methods, the HINT/HINTm interval index has been identified as the superior structure, outperforming the Interval Tree, Timeline Index, 1D-Grid, and Period Index according to recent studies [5]. This indicates a significant advancement in the field of interval indexing, offering more efficient data management and query processing.

2.3 HINT & HINTm

In this subsection, we will review the foundational work upon which this thesis is based, specifically the Hierarchical Index for Intervals (HINT) designed for main memory [5]. It defines a hierarchical domain decomposition and assigns each interval to at most two partitions per level. The primary goal of the index is to minimize the number of comparisons during query evaluation, while keeping the space requirements relatively low, even when there are long intervals in the collection. HINT applies a smart division of intervals in each partition into two groups, which avoids the production and handling of duplicate range query results and minimizes the number of intervals that must be accessed. Here we will describe shortly the first version of the index, HINT, which avoids comparisons overall during query evaluation, but it is not always applicable and may have high space requirements. Subsequently, we will provide a detailed examination of the general version of the index, HINTm, which is applicable to intervals in arbitrary domains. The latest version of the index, along with its optimizations, utilized to developed top k query examination. The following table presents the notations that will be employed throughout the remainder of this research.

Table 2.1 Useful notations

Notation	Description
$s.id, s.st, s.end$	interval id, interval start, interval end
$q.st, q.end$	query start, query end
$prefix(k, x)$	k-bit prefix of integer x
$P_{l,i}$	i-th partition at level l of HINT/HINTm
$P_{l,f}$ ($P_{l,l}$)	first (last) partition at level l that overlaps with q
$P_{l,i}^{O}$ ($P_{l,i}^{R}$)	subpartition of $P_{l,i}$ with originals (replicas)
$P_{l,i}^{Oin}$ ($P_{l,i}^{Oaft}$)	intervals in $P_{l,i}^{O}$ ending inside (after) the partition

2.3.1 HINT

HINT is appropriate in the case of a discrete and not very large domain D. Specifically, assume that the domain D where from the endpoints of intervals in S take value is $[0, 2^m - 1]$. HINT defines a regular hierarchical decomposition of the domain into partitions, where at each level l from 0 to m, there are 2^l partitions, denoted by array $P_{l,0}, \dots, P_{l,2^l-1}$. Figure 2.1 illustrates the hierarchical domain partitioning for $m = 4$.

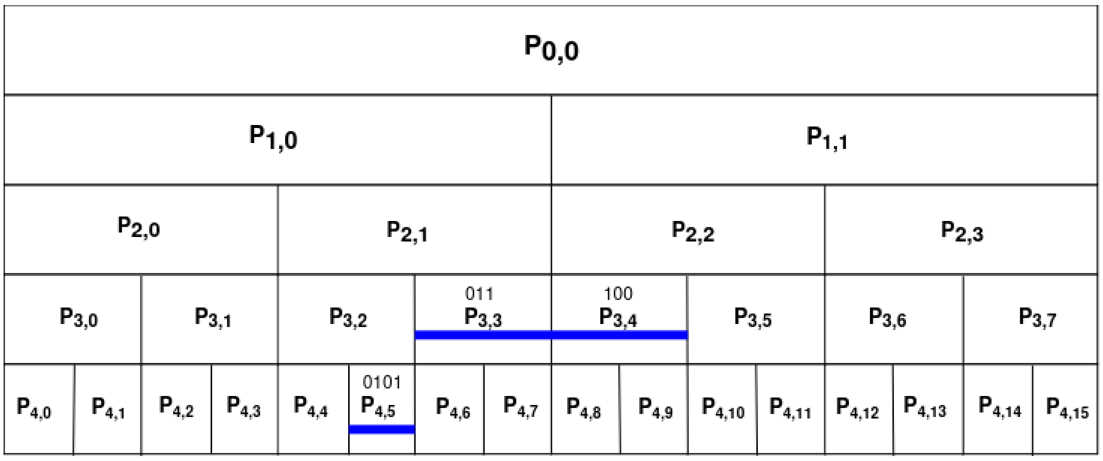
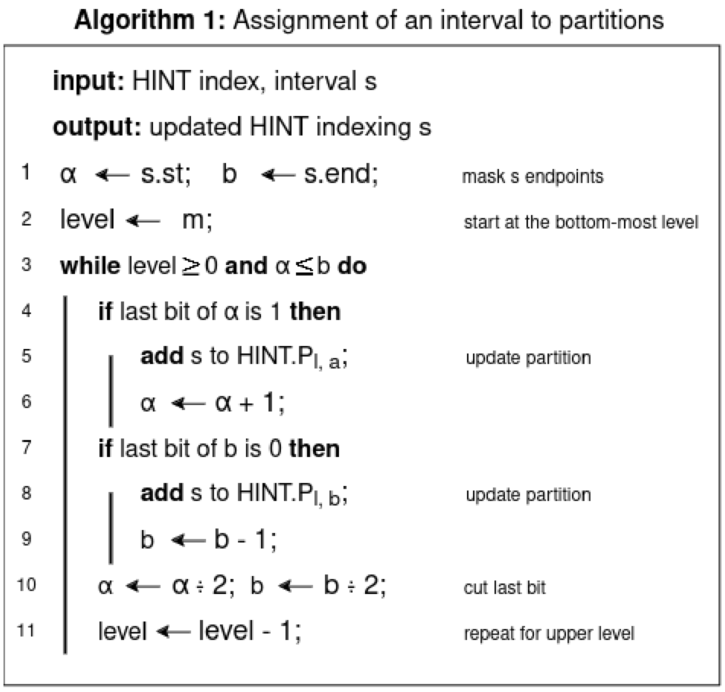


Figure 2.1: Hierarchical domain partitioning for m=4.

Each interval $s \in S$ is assigned to the smallest set of partitions which collectively define s. It is not hard to show that s will be assigned to at most two partitions per level. For example, in Figure 2.1, interval $[5, 9]$ is assigned to one partition at level $l = 4$ and two partitions at level $l = 3$. The assignment procedure is described by Algorithm 1.



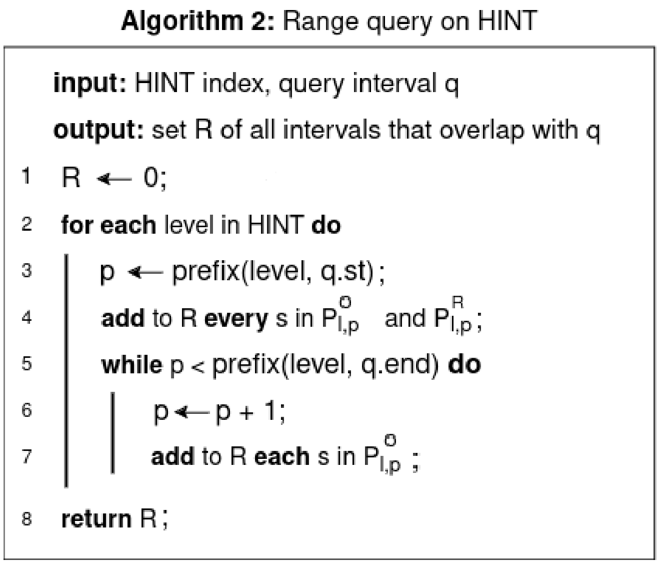
```
Algorithm 1: Assignment of an interval to partitions
input: HINT index, interval s
output: updated HINT indexing s
α ← s.st;   b ← s.end;                  mask s endpoints
level ← m;                              start at the bottom-most level
while level ≥ 0 and α ≤ b do
    if last bit of α is 1 then
        add s to HINT.P_{l, a};         update partition
        α ← α + 1;
    if last bit of b is 0 then
        add s to HINT.P_{l, b};         update partition
        b ← b - 1;
    α ← α ÷ 2; b ← b ÷ 2;               cut last bit
    level ← level - 1;                  repeat for upper level
```

Figure 2.2: Pseudocode for Assigning an Interval to Partitions.

In a nutshell, for an interval $[a, b]$, starting from the bottom-most level l, if the last bit of a (resp. b) is 1 (resp. 0), the interval is assigned to partition $P_{l,a}$ (resp. $P_{l,b}$) and increase a (resp. decrease b) by one. Then a and b are updated by cutting-off their last bits (i.e., integer division by 2, or bitwise right-shift). If, at the next level, $a > b$ holds, indexing $[a, b]$ is done.
The main operation of the index is the execution of range queries. A range query q can be evaluated by finding at each level the partitions that overlap with q. Specifically, the partitions that overlap with the query interval q at level l are partitions $P_{l,prefix(l,q.st)}$ to $P_{l,prefix(l,q.end)}$, where $prefix(n, x)$ denotes the n-bit prefix of integer x. These partitions are called relevant to the query q. All intervals in the relevant partitions are guaranteed to overlap with q and intervals in none of these partitions cannot possibly overlap with q. However, since the same interval s may exist in multiple partitions that overlap with a query, s may be reported multiple times in the query result. For this reason, there is a technique that avoids the production and therefore, the need for elimination of duplicates and, at the same time, minimizes the number of data accesses. For this, the intervals in each partition $P_{l,i}$ are divided into two groups: originals $P^O_{l,i}$ and replicas $P^R_{l,i}$. Group $P^O_{l,i}$ contains all intervals $s \in$

$P_{l,i}$ that begin at $P_{l,i}$ i.e., $prefix(l, s.st) = i$. Group $P^R_{l,i}$ contains all intervals $s \in P_{l,i}$ that begin before $P_{l,i}$ i.e., $prefix(l, s.st) \neq i$. Each interval is added as original in only one partition of HINT. For example, interval $[5, 9]$ in Figure 2.1 is added to $P^O_{4,5}$, $P^R_{3,3}$ and $P^R_{3,4}$.

Given a range query q, at each level l of the index, we report all intervals in the first relevant partition $P_{l,f}$ (i.e., $P^O_{l,f} \cup P^R_{l,f}$). Then, for every other relevant partition $P_{l,i}$, $i > f$, we report all intervals in $P^O_{l,i}$ and ignore $P^R_{l,i}$. This guarantees that no result is missed, and no duplicates are produced. The reason is that each interval s will appear as original in just one partition, hence, reporting only originals cannot produce any duplicates. At the same time, all replicas $P^R_{l,f}$ in the first partitions per level l that overlap with q begin before q and overlap with q, so they should be reported. On the other hand, replicas $P^R_{l,i}$ in subsequent relevant partitions $(i > f)$ contain intervals, which are either originals in a previous partition $P_{l,j}$, $j < i$ or replicas in $P^R_{l,f}$, so, they can safely be skipped. Algorithm 2 describes the range query algorithm using HINT.



Algorithm 2: Range query on HINT

```
input: HINT index, query interval q
output: set R of all intervals that overlap with q
R ← 0;
for each level in HINT do
    p ← prefix(level, q.st);
    add to R every s in $P^O_{l,p}$ and $P^R_{l,p}$;
    while p < prefix(level, q.end) do
        p ← p + 1;
        add to R each s in $P^O_{l,p}$;
return R;
```

Figure 2.3 Pseudocode for Range query on HINT.

For example, consider the hierarchical partitioning of Figure 2.4 and a query interval $[5, 9]$. The binary representations of $q.st$ and $q.end$ are 0101 and 1001, respectively. The relevant partitions at each level are shown in bold (blue) and dashed (red) lines and can be determined by the corresponding prefixes of 0101 and 1001. At each level l, all intervals (both originals and replicas) in the first partitions $P_{l,f}$ (bold/blue) are reported while in the subsequent partitions (dashed/red), only the original intervals are.

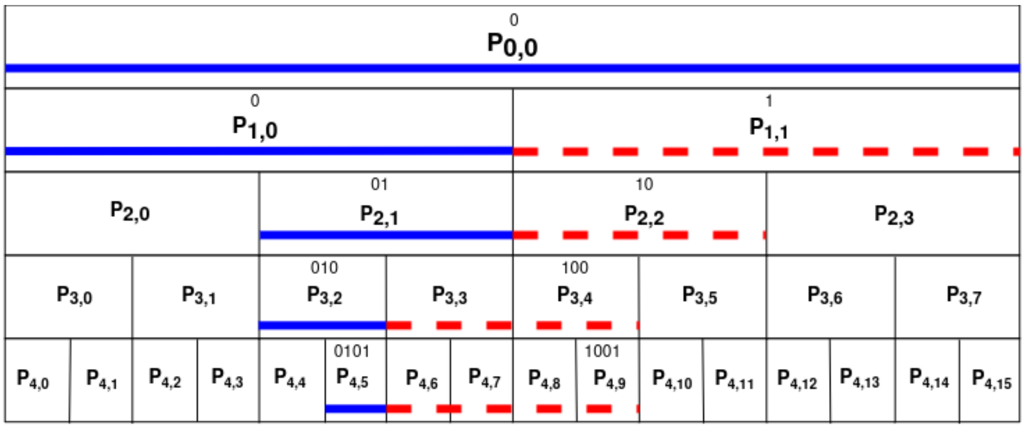


Figure 2.4: Accessed partitions for range query [5, 9].

The version of HINT described above finds all range query results, without conducting any comparisons. This means that in each partition $P_{l,i}$, we only must keep the ids of the intervals that are assigned to $P_{l,i}$ and do not have to store/replicate the interval endpoints. In addition, the relevant partitions at each level are computed by fast bit-shifting operations which are comparison free. To use HINT for arbitrary integer domains, first all interval endpoints should be normalized by subtracting the minimum endpoint, to convert them to values in a $[0, 2^m - 1]$ domain (the same transformation should be applied on the queries).

2.3.2 HINTm

HINTm is used for intervals in arbitrary domains and uses a hierarchical domain partitioning with $m + 1$ levels, based on a $[0, 2^m - 1]$ domain D; each raw interval endpoint is mapped to a value in D, by linear rescaling. The mapping function $f(R \rightarrow D)$ is $f(x) = \frac{x - \min(x)}{\max(x) - \min(x)} * (2^m - 1)$, where $\min(x)$ and $\max(x)$ are the minimum and maximum interval endpoints in the dataset S, respectively. Each raw interval $[s.st, s.end]$ is mapped to interval $[f(s.st), f(s.end)]$. The mapped interval is then assigned to at most two partitions per level in HINTm, using Algorithm 1. For the ease of presentation, assume that the raw interval endpoints take values in $[0, 2^{m'} -$

1], where $m' > m$, which means that the mapping function f simply outputs the m most significant bits of its input. As an example, assume that $m = 4$ and $m' = 6$. Interval $[21, 38] = [0b010101, 0b100110]$ is mapped to interval $[5, 9] = [0b0101, 0b1001]$ and assigned to partitions $P_{4,5}$, $P_{3,3}$ and $P_{3,4}$, as shown in Figure 2.1. So, in contrast to HINT, the set of partitions where an interval s is assigned in HINTm does not define s, but the smallest interval in the $[0, 2^m - 1]$ domain D, which covers s. As in HINT, at each level l, we divide each partition $P_{l,i}$ to $P_{l,i}^O$ and $P_{l,i}^R$, to avoid duplicate query results.

For a range query q, simply reporting all intervals in the relevant partitions at each level (as in Algorithm 2) would produce false positives. Instead, comparisons to the query endpoints may be required for the first and the last partition at each level that overlap with q. Specifically, consider each level of HINTm as a 1D-grid and go through the partitions at each level l that overlap with q.

For the first partition $P_{l,f}$, verify whether s overlaps with q for each interval $s \in P_{l,f}^O$ and each $s \in P_{l,f}^R$. For the last partition $P_{l,l}$, verify whether s overlaps with q for each interval $s \in P_{l,l}^O$. For each partition $P_{l,i}$ between $P_{l,f}$ and $P_{l,l}$, report all $s \in P_{l,i}^O$ without any comparisons. As an example, consider the HINTm index and the range query interval q shown in Figure 2.5.

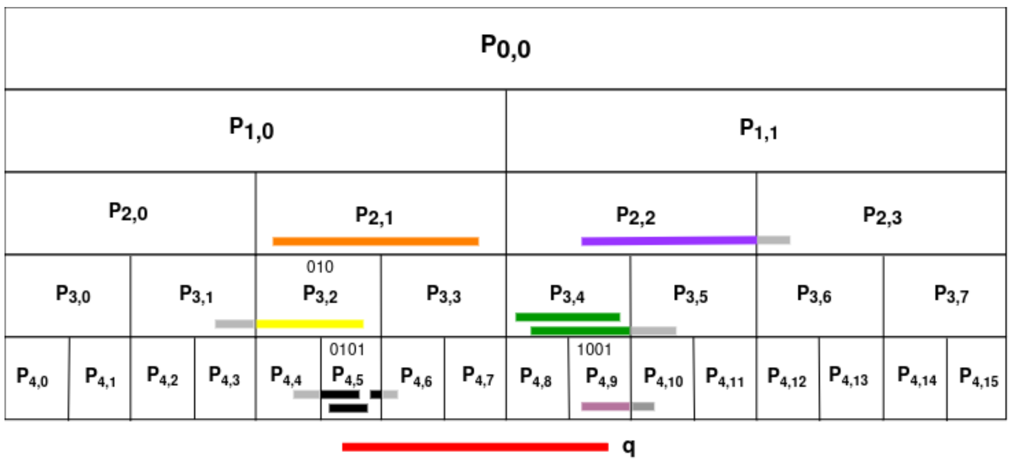
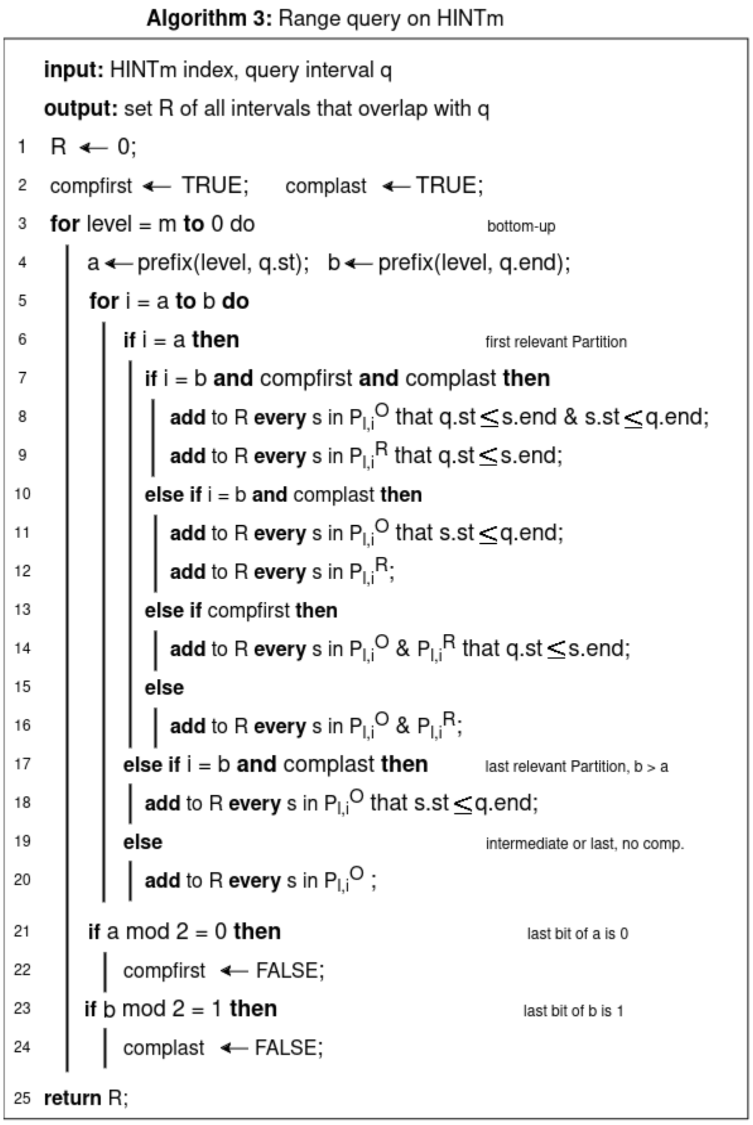


Figure 2.5: Avoiding redundant comparisons on HINTm.

The identifiers of the relevant partitions to q are shown in the Figure 2.5 (and some indicative intervals that are assigned to these partitions). At level $m = 4$, comparisons

must be performed for all intervals in the first relevant partitions $P_{4,5}$. In partitions $P_{4,6}$,..., $P_{4,8}$ we just report the originals in them as results, while in partition $P_{4,9}$ we compare the start points of all originals with q, before confirming whether they are results or not. At the first and the last partition of each level l overlap tests can be simplified based on the following: At every level l, each $s \in P^R_{l,f}$ is a query result $iff\ q.st \leq s.end$. If $l > f$, each $s \in P^O_{l,f}$ is a query result $iff\ s.st \leq q.end$. Resulting from the fact that for the first relevant partition $P_{l,f}$ at each level l, for each replica $s \in P^R_{l,f}, s.st < q.st$, so $q.st \leq s.end$ suffices as an overlap test and for the last partition $P_{l,l}$, if $l > f$, for each original $s \in P^O_{l,f}$, $q.st < s.st$, so $s.st \leq q.end$ suffices as an overlap test.
One of the most important findings in the study and a powerful feature of HINTm is that at most levels, it is not necessary to do comparisons at the first and/or the last partition. For instance, in the previous example, comparisons do not have to be performed for partition $P_{3,4}$, since any interval assigned to $P_{3,4}$ should overlap with $P_{4,8}$ and the interval spanned by $P_{4,8}$ is covered by q. This means that the start points of all intervals in$P_{3,4}$ is guaranteed to be before $q.end$ (which is inside $P_{4,9}$). In addition, observe that for any relevant partition which is the last partition at an upper level and covers $P_{3,4}$ (i.e., partitions $\{P_{2,2}, P_{1,1}, P_{0,0}\}$), we do not have to conduct the $s.st \leq q.end$ tests as intervals in these partitions are guaranteed to start before $P_{4,9}$. The following formalizes these observations: If the first (resp. last) relevant partition for a query q at level l $(l < m)$ starts (resp. ends) at the same value as the first (resp. last) relevant partition at level $l + 1$, then for every first (resp. last) relevant partition $P_{v,f}$(resp. $P_{v,l}$) at levels $v < l$, each interval $s \in P_{v,f}$ (resp. $s \in P_{v,l}$) satisfies $s.end \geq q.st$ (resp. $s.st \geq q.end$). Algorithm 3 is a pseudocode for the range query algorithm on HINTm.



Algorithm 3: Range query on HINTm

```
   input: HINTm index, query interval q
   output: set R of all intervals that overlap with q
1  R ← 0;
2  compfirst ← TRUE;    complast ← TRUE;
3  for level = m to 0 do                                   bottom-up
4      a ← prefix(level, q.st);   b ← prefix(level, q.end);
5      for i = a to b do
6          if i = a then                                   first relevant Partition
7              if i = b and compfirst and complast then
8                  add to R every s in $P_{l,i}^O$ that q.st ≤ s.end & s.st ≤ q.end;
9                  add to R every s in $P_{l,i}^R$ that q.st ≤ s.end;
10             else if i = b and complast then
11                 add to R every s in $P_{l,i}^O$ that s.st ≤ q.end;
12                 add to R every s in $P_{l,i}^R$;
13             else if compfirst then
14                 add to R every s in $P_{l,i}^O$ & $P_{l,i}^R$ that q.st ≤ s.end;
15             else
16                 add to R every s in $P_{l,i}^O$ & $P_{l,i}^R$;
17         else if i = b and complast then                 last relevant Partition, b > a
18             add to R every s in $P_{l,i}^O$ that s.st ≤ q.end;
19         else                                            intermediate or last, no comp.
20             add to R every s in $P_{l,i}^O$ ;
21     if a mod 2 = 0 then                                 last bit of a is 0
22         compfirst ← FALSE;
23     if b mod 2 = 1 then                                 last bit of b is 1
24         complast ← FALSE;
25 return R;
```

Figure 2.6: Pseudocode for Range query on HINTm.

The algorithm accesses all levels of the index, bottom-up. It uses two auxiliary flag variables $compfirst$ and $complast$ to mark whether it is necessary to perform comparisons at the current level (and all levels above it) at the first and the last partition, respectively. At each level l, offsets of the relevant partitions to the query are found, based on the $l - prefixes$ of $q.st$ and $q.end$ (Line 4). For the first position $f = prefix(q.st)$, the partitions holding originals and replicas $P^O_{l,f}$ and $P^R_{l,f}$ are accessed. The algorithm first checks whether $f = l$, i.e., the first and the last partitions coincide. In this case, if $compfirst$ and $complast$ are set, then all comparisons are performed in $P^O_{l,f}$ and apply what described previously in $P^R_{l,f}$. Else, if only $complast$ is set, the $q.st \leq s.end$ comparisons can be safely skipped; if only compfist is set, regardless whether $f = l$, $q.st \leq s.end$ comparisons are performed to both originals and replicas to the first partition. Finally, if neither $compfirst$ nor $complast$ are set, all intervals are just reported in the first partition as results. For the last partition $P_{l,l}$ if $l > f$ (line 17) then $P_{l,l}^{O}$ is examined by just applying the $s.st \leq q.end$ test for each interval there. Finally, for all partitions in-between the first and the last one, all original intervals there are simply reported.

2.3.3 Optimized HINTm

As previously discussed, the primary advantage of HINT/HINTm is its ability to minimize the number of comparisons during the evaluation of a range query. Consequently, for most examined partitions, specifically intermediate partitions, there is no need to access the endpoints of intervals. Instead, only the interval ids are required to report a range query result. This observation led the authors to design certain optimizations that involve retaining only the interval ids.

However, in the context of ranking queries, where it is necessary to determine the rank of a result based on the overlap between the interval and the query, the interval endpoints must be accessed each time. Therefore, optimizations that do not involve accessing the interval endpoints are unsuitable for the evaluation of ranking queries. This subsection will discuss the optimization techniques that applied on the evaluation of range queries and are appropriate for retrieving ranking queries results also. The main method 'Subdivisions and space decomposition' [5] reduces the number of partitions in HINTm where comparisons are performed and avoids accessing unnecessary data. Recall that, at each level l of HINTm, every partition $P_{l,i}$ is divided into ${P_{l,i}}^{O}$ (holding originals) and ${P_{l,i}}^{R}$ (holding replicas). Now the authors propose to further divide each ${P_{l,i}}^{O}$ into ${P_{l,i}}^{Oin}$ and ${P_{l,i}}^{Oaft}$, so that ${P_{l,i}}^{Oin}$ (resp. ${P_{l,i}}^{Oaft}$) holds the intervals from ${P_{l,i}}^{Oin}$ that end inside (resp. after) partition $P_{l,i}$. Similarly, each ${P_{l,i}}^{R}$ is divided into ${P_{l,i}}^{Rin}$ and ${P_{l,i}}^{Raft}$. Consider a range query q, which overlaps with a sequence of more than one partition at level l. As already discussed, if comparisons must be conducted in the first such partition $P_{l,f}$, should be done for all intervals in ${P_{l,f}}^{O}$and ${P_{l,f}}^{R}$. The subdivision of ${P_{l,f}}^{O}$and ${P_{l,f}}^{R}$, concludes to the following: If $P_{l,f} \neq P_{l,l}$ each interval s in ${P_{l,f}}^{Oin} \cup {P_{l,f}}^{Rin}$ overlaps with q iff $s.end \geq q.st$; and all intervals s in ${P_{l,f}}^{Oaft}$and ${P_{l,f}}^{Raft}$ are guaranteed to overlap with q. Follows directly from the fact that q starts inside $P_{l,f}$ but ends after $P_{l,f}$. Hence, just one comparison is needed for each interval in ${P_{l,f}}^{Oin} \cup {P_{l,f}}^{Rin}$, whereas all intervals ${P_{l,f}}^{Oaft} \cup {P_{l,f}}^{Raft}$ can be reported as query results without any comparisons.

As already discussed, for all partitions $P_{l,i}$ between $P_{l,f}$ and $P_{l,l}$, intervals in ${P_{l,i}}^{Oin} \cup {P_{l,i}}^{Oaft}$are just reported as results, without any comparisons, whereas for the last partition $P_{l,l}$, one comparison is performed per interval in ${P_{l,l}}^{Oin} \cup {P_{l,l}}^{Oaft}$. If the range query q overlaps only one partition $P_{l,f}$ at level l, the authors use following to minimize the necessary comparisons: If $P_{l,f} = P_{l,l}$ then each interval s in ${P_{l,f}}^{Oin}$ overlaps with q iff $s.st \leq q.end \wedge q.st \leq s.end$, each interval s in ${P_{l,f}}^{Oaft}$ overlaps with q, iff $s.st \leq q.end$, each interval s in ${P_{l,f}}^{Rin}$ overlaps with q, iff $s.end \geq q.st$, all intervals in ${P_{l,f}}^{Raft}$ overlap with q. All intervals $s \in {P_{l,f}}^{Oaft}$and end after q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in {P_{l,f}}^{Rin}$ start before q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in {P_{l,f}}^{Raft}$ start before and end after q, so they are guaranteed results.

Overall, the subdivisions minimize the number of intervals in each partition, for which we must apply comparisons. Figure 2.7 shows the subdivisions which are accessed by query q at level $l = 2$ of a HINTm index. In partition $P_{l,f} = P_{2,1}$, all four subdivisions are accessed, but comparisons are needed only for intervals in ${P_{2,1}}^{Oin}$ and ${P_{2,1}}^{Rin}$. In partition $P_{2,2}$, only the originals (${P_{2,2}}^{Oin}$ and ${P_{2,2}}^{Oaft}$) are accessed and reported without any comparisons. Finally, in $P_{l,f} = P_{2,3}$, only the originals (${P_{2,3}}^{Oin}$ and ${P_{2,3}}^{Oaft}$) are accessed and compared to q.

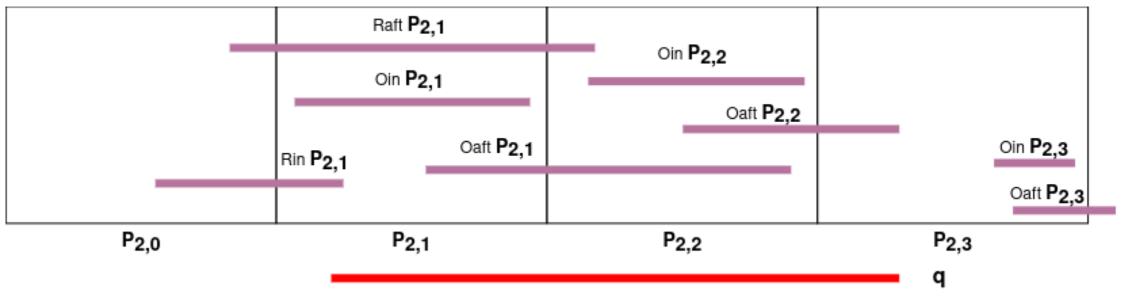


Figure 2.7: Partition subdivisions in HINTm level 2.

As can be easily summarized, the intervals in each subdivision can be kept sorted, to reduce the number of comparisons for queries that access them. For example, by examining the last partition $P_{l,l}$, that overlaps with a query q at a level l. Any can conclude that if the intervals s in ${P_{l,f}}^{Oin}$ are sorted on their start endpoint (i.e., $s.st$), then the intervals can be reported until the first $s \in {P_{l,l}}^{Oin}$, such that $s.st > q.end$. Or binary search can be performed to find the first $s \in {P_{l,l}}^{Oin}$, such that $s.st > q.end$ and then scan and report all intervals before s. Table 2.2 summarizes the sort orders for each of the four subdivisions of a partition that can be beneficial in range query evaluation. For a subdivision ${P_{l,l}}^{Oin}$, intervals may have to be compared based on their start point (if $P_{l,i} = P_{l,f}$), or based on their end point (if $P_{l,i} = P_{l,l}$), or based on both points (if $P_{l,i} = P_{l,f} = P_{l,l}$). Hence, they choose to sort based on either $s.st$ or $s.end$ to accommodate two of these three cases. For a subdivision ${P_{l,i}}^{Oaft}$, intervals may only have to be compared based on their start point (if $P_{l,i} = P_{l,l}$). For a subdivision ${P_{l,i}}^{Rin}$, intervals may only have to be compared based on their end point $P_{l,i} = P_{l,f}$. Last, for a subdivision ${P_{l,i}}^{Raft}$, there is never any need to compare the intervals, so, no order provides any search benefit.

Table 2.2: Beneficial sort orders

Subdivision	Beneficial sorting
${P_{l,i}}^{Oin}$	by $s.st$ or by $s.end$
${P_{l,i}}^{Oaft}$	by $s.st$
${P_{l,i}}^{Rin}$	by $s.end$
${P_{l,i}}^{Raft}$	no sorting

Thus far, we have thoroughly examined all the significant contributions made by Christodoulou et al. in [5]. The subsequent sections of this paper will delve into the definitions, methodologies, and techniques that contribute to the development of HINTm, aiming to efficiently support ranking (top k) queries.

CHAPTER 3

PROBLEM DEFINITIONS

3.1 Top-k Query

In the context of computer science and information retrieval, top k query is a type of query that retrieves the top k items from a dataset based on some specified criteria. The scientific definition encompasses the following key aspects:

- **Definition**: A top k query is an operation that returns the k highest ranking results from a collection of items, based on a given score or a ranking function.
- **Ranking Function**: The ranking function $f(x)$ assigns a numerical score to each item x in the dataset. The function reflects the relevance, similarity, or preference according to the specific application or query context.

3.2 Overlap-Intersection

In mathematics, an interval is a fundamental concept used to describe a continuous range of numbers. There are various types of intervals, this study focuses particularly on closed intervals. A closed interval is a set of real numbers that includes both its endpoints. It is denoted by $[a, b]$, where a and b are the two endpoints of the interval, and all numbers between a and b including a and b themselves, belong to the interval [14].

When considering multiple intervals, a common problem is determining their intersection (overlap). The intersection of two intervals refers to the set of points that are contained within both intervals. For two closed intervals $[a, b]$ and $[c, d]$, their intersection is also a closed interval if they do overlap. Mathematically, the intersection of these intervals is defined as $[\max(a, c), \min(b, d)]$. This intersection is valid and non-empty:

- $iff \max(a, c) \leq \min(b, d)$.

If this condition is not met, the intervals do not overlap, and their intersection is the empty set. For example, consider two closed intervals $[1, 5]$ and $[3, 7]$. The intersection of these intervals would be $[\max(1, 3), \min(5, 7)] = [3, 5]$. This resultant interval includes all numbers that lie within both original intervals.

In this study, we refer to this concept as the absolute overlap or intersection between two intervals, which will determine the ranking function examined. More explanatory, given as data input a set of closed intervals S and a closed interval q as a query. We determine the intersection or overlap of an interval $s \in S$ based on the following:

- **Absolute intersection**: $|s \cap q| = [\max(q.st,\ s.st), \min(q.end, s.end)$.

3.3 Ranking Problem

Given a large collection of intervals S, a positive integer k and a query interval q. We address the problem of finding the top k intervals of the collection S that best overlap with the query q. The ranking score s_{score} is computed by the endpoints of absolute intersection between query q and interval $s \in S$, formulated as:

- **Intersection score**: $s_{score} = \min(q.end,\ s.end) - \max(q.st,\ s.st)$.

- For each interval: $z \in S \; and \; |z \cap q| = \emptyset \Rightarrow z_{score} = 0$.

The goal is to identify the k intervals with the highest overlapping scores, thus ranking them based on how well they intersect with the query. Formally,

Input:

- Finite set of intervals: $S = \{s_1, s_2, \dots, s_n\}$
- Positive integer: k
- Interval query: q

Output:

- Subset: $TOPk \subseteq S$, such that $|TOPk| = k$ and each interval $s \in TOPk$ has one of the k highest s_{score} with q.

The Figure 3.1 below presents a comprehensible example that encapsulates the concepts discussed so far.

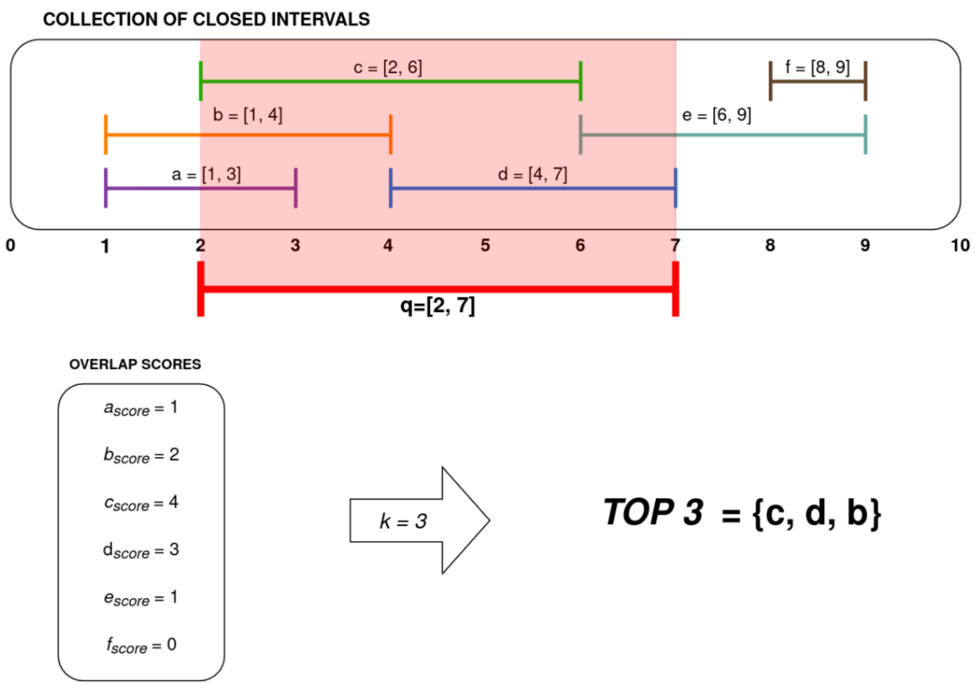


Figure 3.1 Ranking example

CHAPTER 4

TRAVERSING METHODS

The primary objective of this study is to address the interval ranking problem with maximum efficiency. To achieve this, we enhance the HINTm index to support top-k queries. This development incorporates the HINTm index along with the optimizations detailed in the subsection 2.3. To accomplish this, we initially use Algorithm 1 “Assignment of an interval to partitions” (figure 2.2) to index all intervals of S. Subsequently we modify Algorithm 3, “Range Query on HINTm” (figure 2.6) so that each time a range query result s is reported, its intersection score is computed. If the $TOPk$ set is not yet full, the interval is added to the set. Otherwise, its score is compared to the k-th highest score in the $TOPk$ set. If the score meets the necessary threshold, the interval is added to the top-k set; otherwise, the algorithm proceeds to the next range query result. For the remainder of this study, this process will be referred to as "update $TOPk$." To check if the $TOPk$ set is not full, we will use the condition:

- $|TOPk| < k$

It is necessary to indicate that if the query overlaps fewer than k intervals, the results should be output directly. While for tracking the best of the results we are using a min-heap data structure due to its efficient insertion and removal operations. In the rest of this section, we will examine the different methods of traversing the HINTm index aiming to extract the top k results as efficiently as possible.

4.1 Naïve Traversal

The initial approach is a naive method that traverses the index in a bottom-up manner, as exactly described previously, modifying the Algorithm 3 "Range Query on HINTm". Each time the algorithm reports a range query result, the overlap score between the query's endpoints and the result's endpoints is computed. Based on this score, it is then determined whether the result should be included in the $TOPk$ set. This process, "update $TOPk$", continues until all potential range query results have been examined.

Naive: Top k query on HINTm

```
input: HINTm index, query interval q
output: set TOP-k with the k intervals that have the highest score with q
TOP-k ← 0;
compfirst ← TRUE;   complast ← TRUE;
for level = m to 0 do                                      bottom-up
    a ← prefix(level, q.st);   b ← prefix(level, q.end);
    for i = a to b do
        if i = a then                                      first relevant Partition
            if i = b and compfirst and complast then
                update TOP-k for every s in $P_{l,i}^{O}$ that q.st ≤ s.end & s.st ≤ q.end;
                update TOP-k for every s in $P_{l,i}^{R}$ that q.st ≤ s.end;
            else if i = b and complast then
                update TOP-k for every s in $P_{l,i}^{O}$ that s.st ≤ q.end;
                update TOP-k for every s in $P_{l,i}^{R}$;
            else if compfirst then
                update TOP-k for every s in $P_{l,i}^{O}$ & $P_{l,i}^{R}$ that q.st ≤ s.end;
            else
                update TOP-k for every s in $P_{l,i}^{O}$ & $P_{l,i}^{R}$;
        else if i = b and complast then                    last relevant Partition, b > a
            update TOP-k for every s in $P_{l,i}^{O}$ that s.st ≤ q.end;
        else                                               intermediate or last, no comp.
            update TOP-k for every s in $P_{l,i}^{O}$ ;
    if a mod 2 = 0 then                                    last bit of a is 0
        compfirst ← FALSE;
    if b mod 2 = 1 then                                    last bit of b is 1
        complast ← FALSE;
return TOP-k;
```

Figure 4.1: Pseudocode for Naïve Traversal.

4.2 Top-Down Traversal

It is evident that longer intervals are more likely to overlap significantly with the query. In the context of HINTm, partitions closer to the root are larger and therefore tend to contain longer intervals compared to those nearer the bottom. Consequently, the Top-Down approach traverses the HINTm from the root towards the bottom. This method prioritizes potentially higher scores in the ranking and reduces unnecessary insertions as the traversal nears the lower levels of the index. To benefit from the nature of HINTm, adjustments to the *compfirst* and *complast* flags are needed. Specifically, before initiating the top-down scanning to gather the top k results, the index is first scanned bottom-up to determine the level *complevel* of HINTm where the flags compfirst and complast are met. Subsequently, the index is scanned starting from the root, avoiding comparisons until complevel. Comparisons are then performed from that level until the bottom.

Top-down: Top k query on HINTm

```
input: HINTm index, query interval q
output: set TOP-k with the k intervals that have the highest score with q
TOP-k ← 0;
compfirst ← TRUE;  complast ← TRUE;
for level = m to 0 do                                  bottom-up to find complevel
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    if compfirst or complast then
        if a mod 2 = 0 then                            last bit of a is 0
            compfirst ← FALSE;
        if b mod 2 = 1 then                            last bit of b is 1
            complast ← FALSE;
    else
        complevel ← level;
for level = 0 to complevel do                          top-down until complevel no comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a to b do
        if i = a then                                  first relevant Partition
            update TOP-k for every s in $P_{l,i}^O$ & $P_{l,i}^R$;
        else                                           intermediate or last relevant Partition
            update TOP-k for every s in $P_{l,i}^O$ ;
for level = complevel to m do                          top-down until bottom with comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a to b do
        if i = a then                                  first relevant Partition
            if i = b then
                update TOP-k for every s in $P_{l,i}^O$ that $q.st \leq s.end$ & $s.st \leq q.end$;
                update TOP-k for every s in $P_{l,i}^R$ that $q.st \leq s.end$;
            else
                update TOP-k for every s in $P_{l,i}^O$ & $P_{l,i}^R$ that $q.st \leq s.end$;
        else if i = b and                              last relevant Partition
            update TOP-k for every s in $P_{l,i}^O$ that $s.st \leq q.end$;
        else                                           intermediate relevant no comp.
            update TOP-k for every s in $P_{l,i}^O$ ;
return TOP-k;
```

Figure 4.2 Pseudocode for Top-Down Traversal.

4.3 Upper Bounds

To further expedite the retrieval of the top k intervals, we introduce Upper bounds that serve to eliminate unnecessary scans of partitions. They are used when it is guaranteed that the partition of HINTm to be scanned does not contain any intervals that could be part of the $TOPk$ set. This approach consequently reduces redundant computations and score comparisons. Upper bounds are applied to each partition of the index, indicating the maximum potential interval that it may contain. Subsequently, the intersection score between this potential maximum interval and the query is calculated, establishing the upper bound for our approach.
To ensure no potential results are not skipped, Upper bounds are employed only after the $TOPk$ set is fully populated. If the set is not yet full, any result will be added regardless of its score. Thus, before utilizing upper bounds, it is necessary to verify that the $TOPk$ set is full. If so, the Upper bound is compared with the k-th score. The partition is scanned only if the Upper bound exceeds the k-th score. Formally, assume s' possible maximum interval a partition P can offer.

- We define as static Upper bound: $Ub.P = s'_{score}$

- Pruning condition whether to scan the P:
  $if\ (|TOPk| < k)||(Ub.P > k_{th-score}) \Rightarrow scan\ P$

In the subsequent analysis, we evaluate two distinct versions of the Upper bounds. The initial version employs the endpoints of the partitions ($P.st,\ P.end,\ P.extent = P.end - P.st$) along with those of the queries ($q.st, q.end$) to compute static bounds for each subdivision of a partition accessible during a top-k query. The revised version utilizes metadata to further refine and tighten the upper bounds.

4.3.1 Static Upper Bounds

As discussed in the section 2.3 for a first relevant partition P_f , both originals and replicas should be accessed. We introduce the static upper bounds, $Ub.P_f^{Oin}$, $Ub.P_f^{Oaft}$, $Ub.P_f^{Rin}$, $Ub.P_f^{Raft}$, for each subdivision.

- An Original in subdivision contains intervals that begin and end in this partition. The potential maximum intersection score that a first relevant Original in can provide is: $Ub.P_f^{Oin} = P.end - q.st$ figure 4.3.
- A Replica in subdivision consists of intervals that begin before this partition and end inside it. The estimated maximum intersection score that a Replica in on a first relevant can provide is: $Ub.P_f^{Rin} = P.end - q.st$ figure 4.3.

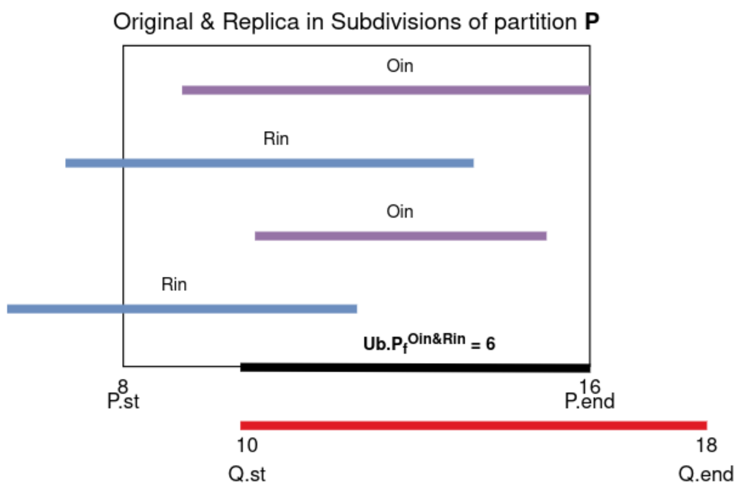


Figure 4.3: Upper Bound of First rel. Partition for Oin & Rin.

- An Original after subdivision contains intervals that begin in this partition but end after. The potential maximum intersection score that an Original after on a first relevant can provide is: $Ub.P_f^{Oaft} = q.end - q.st$ figure 4.4.
- A Replica after subdivision consists of intervals that begin before this partition and end after it also. The hypothetical maximum intersection score that a Replica after on a first relevant partition can provide is: $Ub.P_f^{Raft} = q.end - q.st$ figure 4.4.

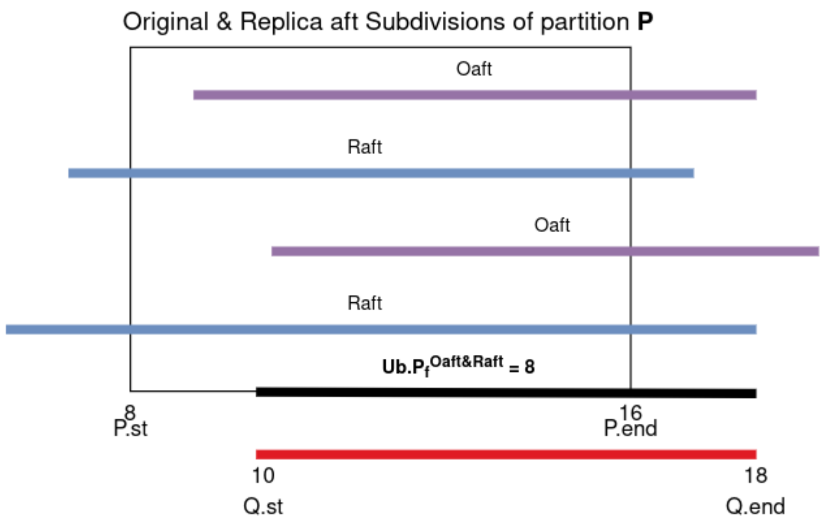


Figure 4.4: Upper Bound of First rel. Partition for Oaft & Raft.

For an intermediate P_{in} or a last relevant P_l partition only originals must be accessed section2.3. We introduce the static upper bounds, $Ub.P_{in}^{Oin}$, $Ub.P_{in}^{Oaft}$, $Ub.P_l^{Oin}$, $Ub.P_l^{Oaft}$, for these cases.

- The possible maximum intersection score that an Original in on an intermediate relevant partition can provide is: $Ub.P_{in}^{Oin} = P.extent$ figure 4.5.

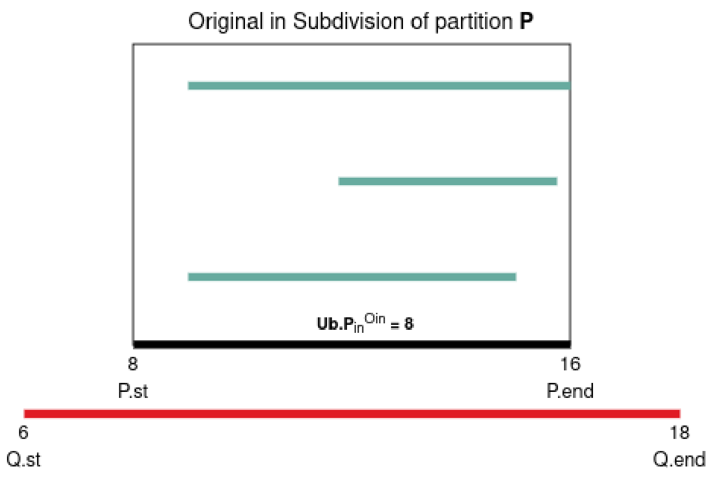


Figure 4.5: Upper Bound of Intermediate rel. Partition for Oin.

- The potential maximum intersection score that an Original after on an intermediate relevant partition can provide is: $Ub.P_{in}^{Oaft} = P.extent + q.end - P.end$.figure 4.6.

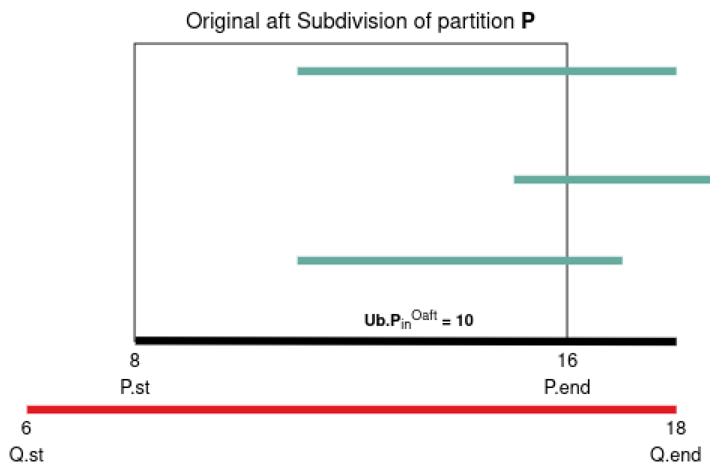


Figure 4.6: Upper Bound of Intermediate rel. Partition for Oaft.

- The likely maximum intersection score that an Original in on a last relevant partition can provide is: $Ub.P_l^{Oin} = q.end - P.st$ figure 4.7.
- The prospective maximum intersection that an Original after on a last relevant partition can provide is: $Ub.P_l^{Oaft} = q.end - P.st$ figure 4.7.

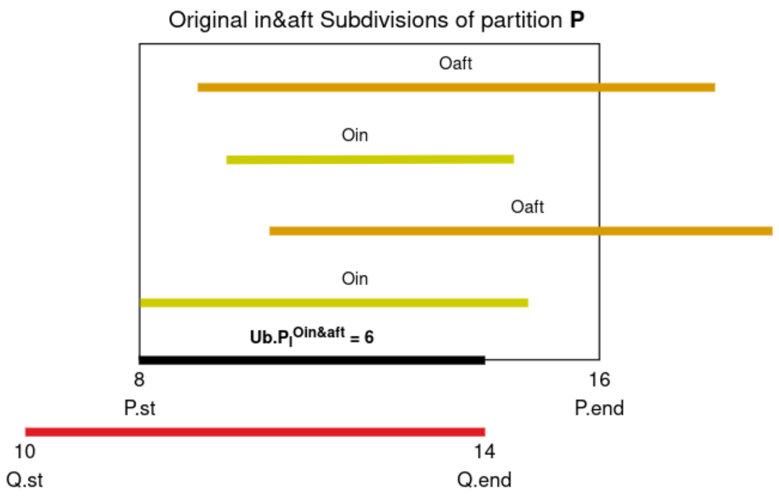


Figure 4.7: Upper Bound of last rel. Partition for Oin & Oaft.

4.3.2 Metadata Upper Bounds

In the revised methodology for determining the Upper bounds, rather than presupposing the expected extended interval that each subclass might offer, we save the longest interval present within each subdivision. This preserved metadata is then utilized to compute the upper bounds. The tracking of the maximum interval for each subdivision is conducted concurrently with the assignment of intervals to their respective partitions. Formally, assume the longest interval $\max s$ that exists on a partition P:

- We define metadata upper bound: $Ub.P_{max} = \max s_{score}$

In such manner, we introduce the metadata upper bounds for every subdivision of a partition P, $Ub.P^{Oin}_{max}$, $Ub.P^{Oaft}_{max}$, $Ub.P^{Rin}_{max}$, $Ub.P^{Raft}_{max}$. In an effort to further tighten the upper bounds, while keeping in mind that the minimum bound is the more suitable. It is important to acknowledge that the superiority of the metadata bounds over the static ones remains uncertain in some cases. Consequently, the implementation will integrate both metadata and static upper bounds in the following manner.

For the first relevant partitions:

- Originals in $upper\ bound^{Oin}_{f} = \min(\mathrm{Ub}.P^{Oin}_{f}, U.P^{Oin}_{max})$, minimum between the static and the metadata bound.
- Originals after $upper\ bound^{Oaft}_{f} = \min\left(\mathrm{Ub}.P^{Oaft}_{f}, U.P^{Oaft}_{max}\right)$, minimum between the static and metadata upper bound.
- Replicas $upper\ bound^{Rin(Raft)}_{f} = Ub.P^{Rin}_{f}(Ub.P^{Raft}_{f})$, static bounds.

For the intermediate relevant partitions:

- Originals in $upper\ bound^{Oin}_{in} = Ub.P^{Oin}_{max}$, metadata bound.
- Originals after $upper\ bound^{Oaft}_{in} = Ub.P^{Oaft}_{l}$, static bound.

For the last relevant partitions:

- Originals in $upper\ bound^{Oin}_{l} = \min(\mathrm{Ub}.P^{Oin}_{l}, U.P^{Oin}_{max})$ minimum between the static and the metadata upper bound.
- Originals after $upper\ bound^{Oaft}_{l} = Ub.P^{Oaft}_{l}$, static upper bound.

In the remainder of this section, we update the naive and top-down methods introduced in the preceding subsections 4.1 and 4.2. Furthermore, we explore novel approaches for traversing the index. This time, however, we employ the discussed Upper bounds to eliminate redundant scans of partitions. The table 4 summarizes the Upper bounds discussed so far.

Table 4.1: Static & Metadata Upper Bounds for relevant partitions

Rel. partition & subdivision	Static Upper Bounds	Metadata Upper Bounds
First rel. Oin	$Ub.P_f^{Oin} = P.end - q.st$	$\min(\mathrm{Ub}.P_f^{Oin}, U.P_{max}^{Oin})$
First rel. Oaft	$Ub.P_f^{Oaft} = q.end - q.st$	$\min\left(\mathrm{Ub}.P_f^{Oaft}, U.P_{max}^{Oaft}\right)$
First rel. Rin	$Ub.P_f^{Rin} = P.end - q.st$	$Ub.P_f^{Rin}$
First rel. Raft	$Ub.P_f^{Raft} = q.end - q.st$	$Ub.P_f^{Raft}$
Intermediate rel. Oin	$Ub.P_{in}^{Oin} = P.extent$	$Ub.P_{max}^{Oin}$
Intermediate rel. Oaft	$Ub.P_{in}^{Oaft} = P.ex + q.end - P.end$	$Ub.P_{in}^{Oaft}$
Last rel. Oin	$Ub.P_l^{Oin} = q.end - P.st$	$\min(\mathrm{Ub}.P_l^{Oin}, U.P_{max}^{Oin})$
Last rel. Oaft	$Ub.P_l^{Oaft} = q.end - P.st$	$Ub.P_l^{Oaft}$

4.4 Naïve & Top-Down with Upper Bounds

In this subsection, we refine the Naive and Top-Down methods by integrating the Upper bounds discussed previously. This enhancement aims to optimize the traversal process by pruning unnecessary partitions. Through this approach, we seek to improve the efficiency and effectiveness of our methods. Incorporating the Upper bounds into the Naïve and Top-Down traversal methods is a straightforward process. The primary requirement is to apply the pruning condition delineated in the previous subsection each time the methods are poised to scan a partition. Specifically, immediately prior to calling "update $TOPk$" in both the Naive and Top-Down methods, the appropriate Upper bound is applied, depending on the subdivision of the partition and its relevance position. This ensures that only those partitions which meet the established criteria are further examined, thereby streamlining the search process.

Explanatory, for example, before scanning the first relevant partition $P_f^{Oin}$of the original in subclass, we apply the following pruning condition:

- $if\ (|TOPk| < k)||(Ub.P_f^{Oin} > k_{score}^{th})\ \Rightarrow scan\ P_f^{Oin}$

By implementing this condition before every partition's scanning of the methods, we establish their optimized versions: Naive with Upper bounds and Top-Down with Upper bounds.

4.5 Depth-First Traversal

Continuing the endeavor to prioritize the longer intervals that populate the index first, as initiated in the Top-Down subsection, we introduce a novel method that traverses the index in a depth-driven manner. This technique diverges from traditional approaches by adopting a more intricate scanning process. Initially, we conduct a top-down traversal, covering only the first relevant partitions. Following this, we examine, in the same way, all the intermediate partitions and finally we scan all the last relevant partitions. The figure 4.8 below shows the relevant partitions that are traversed top-down (first blue, then green and last yellow).

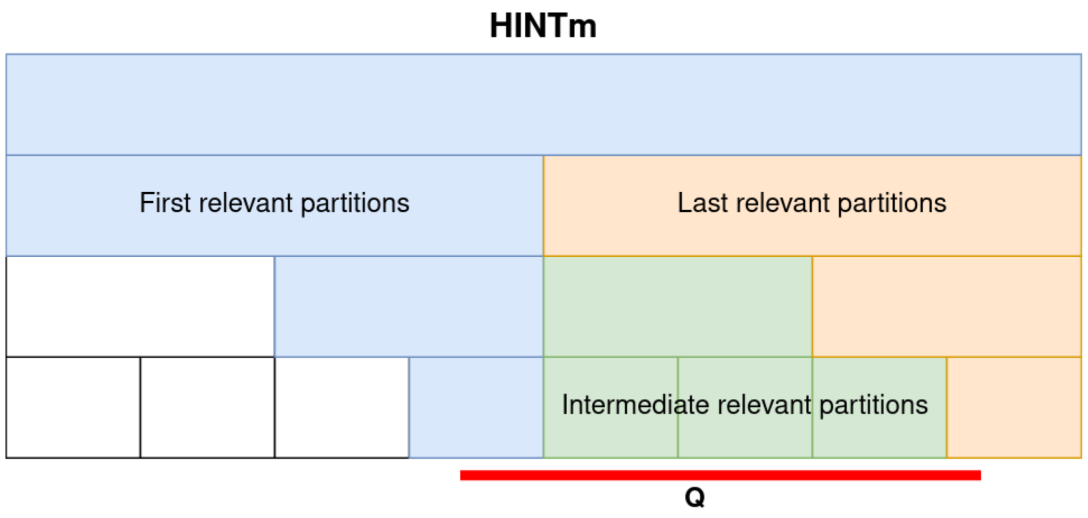


Figure 4.8: First, Intemediate & Last relevant Partitions.

Furthermore, we incorporate the *complevel* parameter to strategically bypass unnecessary comparisons, mirroring the optimization techniques employed in the Top-Down method. Additionally, we apply the upper bounds discussed previously to every call of the $TOPk$ update.

Depth-first traversal: Top k query on HINTm

```
input: HINTm index, query interval q
output: set TOP-k with the k intervals that have the highest score with q
TOP-k ← 0;
compfirst ← TRUE;  complast ← TRUE;
for level = m to 0 do                                        bottom-up to find complevel
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    if compfirst or complast then
        if a mod 2 = 0 then                                  last bit of a is 0
            compfirst ← FALSE;
        if b mod 2 = 1 then                                  last bit of b is 1
            complast ← FALSE;
    else
        complevel ← level;
for level = 0 to complevel do                                top-down first rel. until complevel no comp.
    a ← prefix(level, q.st);
    if (TOP-k is not full) || ($Ub.P_{l,a}^{O\&R}$ > k-th$_{score}$) then      Upper bound condition
        update TOP-k for every s in $P_{l,a}^{O}$ & $P_{l,a}^{R}$;
for level = complevel to m do                                top-down first rel. until bottom with comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    if a = b and then
        if (TOP-k is not full) || ($Ub.P_{l,a}^{O}$ > k-th$_{score}$) then    Upper bound condition
            update TOP-k for every s in $P_{l,a}^{O}$ that q.st ≤ s.end & s.st ≤ q.end;
        if (TOP-k is not full) || ($Ub.P_{l,a}^{R}$ > k-th$_{score}$) then    Upper bound condition
            update TOP-k for every s in $P_{l,a}^{R}$ that q.st ≤ s.end;
    else
        if (TOP-k is not full) || ($Ub.P_{l,a}^{O\&R}$ > k-th$_{score}$) then  Upper bound condition
            update TOP-k for every s in $P_{l,a}^{O}$ & $P_{l,a}^{R}$ that q.st ≤ s.end;
for level = 0 to m do                                        top-down intermediate rel. until bottom no comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a+1 to b-1 do
        if (TOP-k is not full) || ($Ub.P_{l,i}^{O}$ > k-th$_{score}$) then    Upper bound condition
            update TOP-k for every s in $P_{l,i}^{O}$ ;
for level = 0 to complevel do                                top-down last rel. until complevel no comp.
    b ← prefix(level, q.end);
    if (TOP-k is not full) || ($Ub.P_{l,b}^{O}$ > k-th$_{score}$) then        Upper bound condition
        update TOP-k for every s in $P_{l,b}^{O}$ ;
for level = complevel to b do                                top-down last rel. until bottom with comp.
    b ← prefix(level, q.end);
    if (TOP-k is not full) || ($Ub.P_{l,b}^{O}$ > k-th$_{score}$) then        Upper bound condition
        update TOP-k for every s in $P_{l,b}^{O}$ that s.st ≤ q.end;
return TOP-k;
```

Figure 4.9: Pseudocode for Depth-First Traversal.

4.6 Ordered Traversal

In our quest to further enhance the efficiency of index traversal, we present a novel method termed the Ordered Traversal. This technique focuses on scanning the partitions of the index in a specific, strategic order to optimize the retrieval process. By systematically organizing the sequence in which partitions are examined, we aim to maximize the likelihood of quickly identifying intervals that overlap greater with the query while minimizing unnecessary computations. This method builds upon the principles (*complevel* and Upper Bounds) outlined in previous sections. Through this ordered approach, we strive to achieve a more streamlined and effective data management solution.

After identifying, with a bottom-up traverse what the *complevel* is. The process of this method unfolds as follows. The index is scanned in a top-down fashion, while leveraging the upper bounds to enhance efficiency. The scanning procedure, starting from the root, is meticulously ordered in distinct stages:

- First, we scan all the Originals and Replicas after for the first relevant partitions figure 4.10.

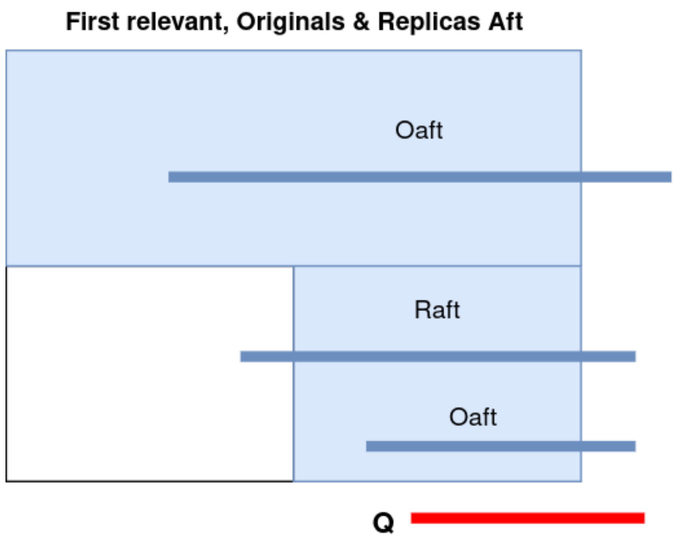


Figure 4.10: First relevant partitions Originals and Replicas after

- Second, we examine all the Originals after for the intermediate relevant partitions 4.8.

- Third, we inspect all the Originals in for the intermediate relevant partitions figure 4.11.

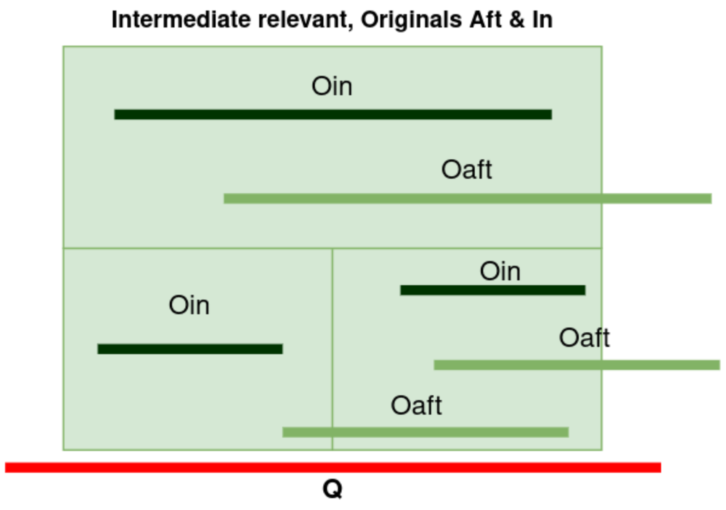


Figure 4.11: Intermediate relevant partitions Originals and Replicas after.

- Fourth, we look at all the Originals and Replicas in for the first relevant partitions figure 4.12.

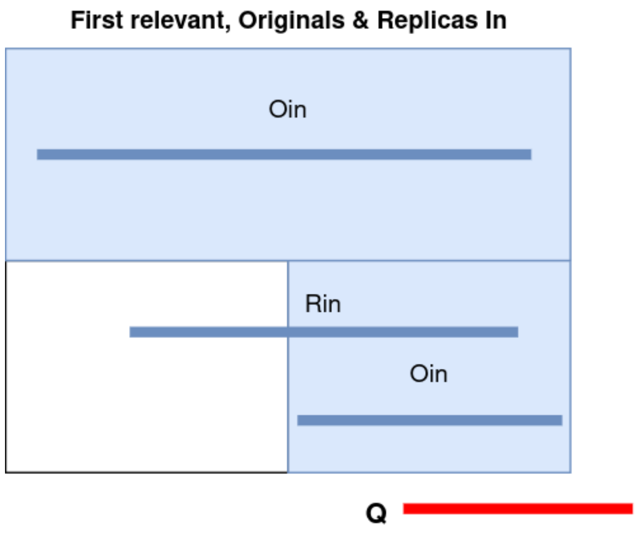


Figure 4.12: First relevant partitions Originals and Replicas in.

- Fifth, we scan the Originals in and after for the last relevant partitions figure 4.13.

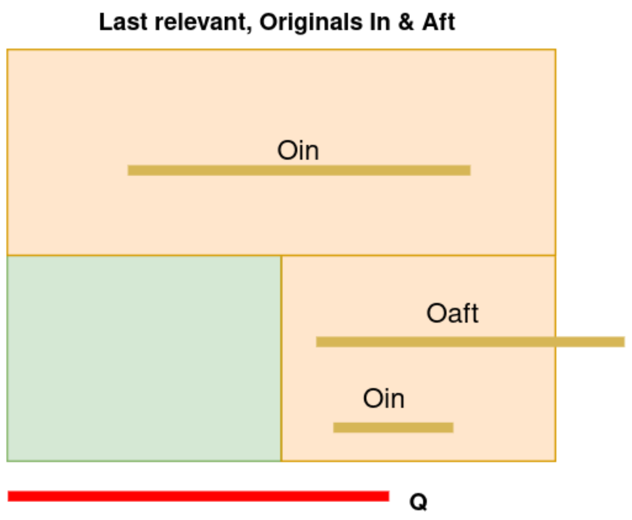


Figure 4.13: Last relevant partitions Originals in and after.

By following this specific order, the method ensures that the most promising partitions are prioritized, aiming to improve the overall effectiveness of the scanning process. Each stage of the scan is carefully designed to build upon the previous one, targeting a comprehensive yet efficient traversal of the index.

Ordered scanning traversal: Top k query on HINTm

```
input: HINTm index, query interval q
output: set TOP-k with the k intervals that have the highest score with q
TOP-k ← 0;
compfirst ← TRUE;  complast ← TRUE;
for level = m to 0 do                                   bottom-up to find complevel
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    if compfirst or complast then
        if a mod 2 = 0 then                             last bit of a is 0
            compfirst ← FALSE;
        if b mod 2 = 1 then                             last bit of b is 1
            complast ← FALSE;
    else
        complevel ← level;
for level = 0 to complevel do                           top-down first rel. Oaft&Raft until complevel no comp.
    a ← prefix(level, q.st);
    if (TOP-k is not full) || (Ub.$P_{l,a}^{Oaft\&Raft}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,a}^{Oaft}$ & $P_{l,a}^{Raft}$;
for level = complevel to m do                           top-down first rel. Oaft&Raft until bottom with comp.
    a ← prefix(level, q.st);
    if (TOP-k is not full) || (Ub.$P_{l,a}^{Oaft}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,a}^{Oaft}$ that s.st ≤ q.end;
    if (TOP-k is not full) || (Ub.$P_{l,a}^{Raft}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,a}^{Raft}$ that;
for level = 0 to m do                                   top-down intermediate rel. Oaft until bottom no comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a+1 to b-1 do
        if (TOP-k is not full) || (Ub.$P_{l,i}^{Oaft}$ > k-$th_{score}$) then     Upper bound condition
            update TOP-k for every s in $P_{l,i}^{Oaft}$ ;
for level = 0 to m do                                   top-down intermediate rel. Oin until bottom no comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a+1 to b-1 do
        if (TOP-k is not full) || (Ub.$P_{l,i}^{Oin}$ > k-$th_{score}$) then     Upper bound condition
            update TOP-k for every s in $P_{l,i}^{Oin}$;
for level = 0 to complevel do                           top-down first rel. Oin&Rin until complevel no comp.
    a ← prefix(level, q.st);
    if (TOP-k is not full) || (Ub.$P_{l,a}^{Oin\&Rin}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,a}^{Oin}$ & $P_{l,a}^{Rin}$;
for level = complevel to m do                           top-down first rel. Oin&Rin until bottom with comp.
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    if a = b and then
        if (TOP-k is not full) || (Ub.$P_{l,a}^{Oin}$ > k-$th_{score}$) then     Upper bound condition
            update TOP-k for every s in $P_{l,a}^{Oin}$ that q.st ≤ s.end & s.st ≤ q.end;
        if (TOP-k is not full) || (Ub.$P_{l,a}^{Rin}$ > k-$th_{score}$) then     Upper bound condition
            update TOP-k for every s in $P_{l,a}^{Rin}$ that q.st ≤ s.end;
    else
        if (TOP-k is not full) || (Ub.$P_{l,a}^{Oin\&Rin}$ > k-$th_{score}$) then     Upper bound condition
            update TOP-k for every s in $P_{l,a}^{Oin}$&$P_{l,a}^{Rin}$ that q.st ≤ s.end;
for level = 0 to complevel do                           top-down last rel. Oin&aft until complevel no comp.
    b ← prefix(level, q.end);
    if (TOP-k is not full) || (Ub.$P_{l,b}^{O}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,b}^{O}$ ;
for level = complevel to b do                           top-down last rel. Oin&aft until bottom with comp.
    b ← prefix(level, q.end);
    if (TOP-k is not full) || (Ub.$P_{l,b}^{O}$ > k-$th_{score}$) then     Upper bound condition
        update TOP-k for every s in $P_{l,b}^{O}$ that s.st ≤ q.end;
return TOP-k;
```

Figure 4.14: Pseudocode for Ordered Traversal.

4.7 Sorted Traversal

In our ongoing effort to optimize index traversal, we introduce the Sorted Traversal of the HINTm. This method takes a comprehensive approach by first gathering all the subclasses of the relevant partitions and then sorting them based on their Upper bounds. By organizing the subdivisions of the partitions in this manner, we can prioritize the most promising candidates for efficient scanning. This technique leverages the previously discussed Upper bound principles.
The process of this method consists of three distinct stages. In the first stage, the index is scanned in a bottom-up fashion (like Naïve Traversal), during which every subclass of the relevant partitions is gathered to a list. The second stage involves sorting these subclasses in descending order according to their upper bounds. This sorting process ensures that the partitions with the highest potential intersections are prioritized. In the final stage, the list of subclasses are scanned sequentially, starting from the subdivision with the greatest Upper bound, and continuing until the following termination condition is satisfied.
More specifically, each time a sub-partition is scanned, it is necessary to verify whether the top-k set is complete. Subsequently, if the k-th score within the top-k set is greater than or equal to the Upper bound of the next sub-partition for scanning, this condition ensures that no other subclass can provide a superior interval result for the $TOPk$ set. If this criterion is met, it is guaranteed that further scanning of remaining subclasses will not yield better results for the $TOPk$ set, thus allowing the process to terminate efficiently. Formally, assume that P' the partition that is about to get scanned:

- $if\ (|TOPk| = k)\ and\ (Ub.P' < k^{th}_{score})\ then\ terminate\ process.$

This methodical approach guarantees that the most significant partitions are examined first.

Sorted scanning traversal: Top k query on HINTm

```
input: HINTm index, query interval q
output: set TOP-k with the k intervals that have the highest score with q
TOP-k ← 0;
compfirst ← TRUE;  complast ← TRUE;
PartitionsToScan ← 0;
for level = m to 0 do                                      bottom-up
    a ← prefix(level, q.st);  b ← prefix(level, q.end);
    for i = a to b do
        if i = a then                                      first relevant Partition
            if i = b and compfirst and complast then
                add $P_{l,i}^{O}$ to PartitionsToScan;
                add $P_{l,i}^{R}$ to PartitionsToScan;
            else if i = b and complast then
                add $P_{l,i}^{O}$ to PartitionsToScan;
                add $P_{l,i}^{R}$ to PartitionsToScan;
            else if compfirst then
                add $P_{l,i}^{O}$ & $P_{l,i}^{R}$ to PartitionsToScan;
            else
                add $P_{l,i}^{O}$ & $P_{l,i}^{R}$ to PartitionsToScan;
        else if i = b and complast then                    last relevant Partition, b > a
            add $P_{l,i}^{O}$ to PartitionsToScan;
        else                                               intermediate or last, no comp.
            add $P_{l,i}^{O}$ to PartitionsToScan;
    if a mod 2 = 0 then                                    last bit of a is 0
        compfirst ← FALSE;
    if b mod 2 = 1 then                                    last bit of b is 1
        complast ← FALSE;
sort PartitionsToScan;                                     based on their upper bounds
for P in PartitionsToScan                                  start from partition with greater upper bound
    update TOP-k for every s in P;                         with respect to comp.
    if (TOP-k is full) and ($Ub.P_{next}$ < k-$th_{score}$) then       termination continion
        break;
return TOP-k;
```

Figure 4.15: Pseudocode for Sorted Traversal

CHAPTER 5

Experimental Evaluation

We employed a system featuring an Intel Core i5 7200U CPU, equipped with 2 cores and 4 threads, operating at a clock rate of 2.5 GHz. The system was configured with 8 GB of RAM. For the operating system, we utilized Linux Ubuntu 22.04.4 in 64-bit mode, and the code was compiled using GCC version 11.4.0.

5.1 Datasets & Queries

The experimental analysis utilized four distinct datasets representing various real-time intervals. The first dataset, referred to as "BOOKS," (https://www.odaa.dk) encompasses time intervals corresponding to instances when books were borrowed from Aarhus libraries in 2013 and contains relatively large intervals. The second dataset, "TAXIS"(https://www.nyc.gov/site/tlc/index.page), includes shorter time periods denoting the pick-up and drop-off times of taxi trips within New York City during a specific period in 2013. Additionally, we conducted experiments using the "BIKES” (https://citibikenyc.com/), which refers to the time spans during which bicycles were rented in New York City in 2020, this dataset has similar characteristics with the “TAXIS”. Lastly, the "FIRES” (https://www.fs.usda.gov/rds/archive/catalog/RDS-2013-0009.4) dataset includes time intervals corresponding to instances of wildfires occurring in the United States between 1992 and 2015. The particular dataset contains lesser intervals but the average duration of them is between the interval duration of the other datasets.

Table 5.1: Dataset characteristics

	BOOKS	TAXIS	BIKES	FIRES
Number of intervals	2.050.707	43.167.001	19.474.352	778.410
Domain size [sec]	31.413.600	31.542.251	31.947.359	757.382.940
Min. duration [sec]	1	1	1	1
Max. duration [sec]	31.406.400	2.148.385	3.786.188	9.988.800
Avg. duration [%]	6,98	0,0024	0,0041	0,013
m [index’s levels]	10	17	16	16

As interval queries, we employed predefined percentage intervals relative to the domain size of individual datasets. Specifically, the query intervals were set at 0,01%, 0,05%, 0,1%, 0,5%, and 1%. Each dataset underwent straightforward testing with 10.000 randomized queries per percentage interval, focusing on measuring total throughput expressed in queries per second. This approach was chosen deliberately over assessing average query time, particularly in contexts where large volumes of interval data are processed.

5.2 HINTm & Methods

The methods described in the section 4 were built on the top of HINTm index, enhanced with the optimizations discussed in the subsection 2.3. For the m parameter (table 5.1), the optimal value was used, determined automatically for each dataset by the already existing code of HINTm. The detailed process for determining the optimal m value was explained in the prior work [5].

All methods were tested on each dataset. We distinguished the methods based on whether the Upper Bounds (static or metadata subsection 4.3) were applied. The table 5.2 below summarizes the methods of our experiments.

Table 5.2: Top-k Querying Methods examined.

without upper bounds	with static upper bounds	with metadata upper bounds
Naive	Naive	Naive
Top-Down	Top-Down	Top-Down
-	Depth-First	Depth-First
-	Ordered Traversal	Ordered Traversal
-	Sorted Traversal	Sorted Traversal

5.3 Query Extent

In the first phase of our experimental evaluation, we focused on exploring the impact of query extent variation through top-k queries across the datasets. Each dataset was subjected to top-k queries representing a range of domain coverage percentages 5.2, thereby investigating how query scope influences system performance. Throughout these experiments, we maintained a constant value of k, set at 10, to ensure consistency in methods' complexity and evaluation metrics. The following subsection presents a detailed analysis of our findings, highlighting the outcomes observed across varying query extents.

For the datasets "TAXIS", "BIKES", and "FIRES", which feature relatively small duration intervals, we observed a high throughput, while decreasing, across query extents ranging from 0,01% to 0,1% of the domain size. However, beyond this range, there is a noticeable decline in throughput. In contrast, the "BOOKS" dataset, characterized by larger duration intervals, exhibited consistently low throughput across all query extents tested. Moreover, we noted that applying static upper bounds accelerated query processing across “TAXIS”, “BIKES” and “FIRES”. For the dataset of “BOOKS” the outcomes remain at almost the same levels. We use the x-axis to represent query extent (fixed percentage of the domain) and the y-axis for throughput, employing a logarithmic scale for "TAXIS" and "BIKES" to accurately depict the performance. Here, we applied static Upper Bounds.

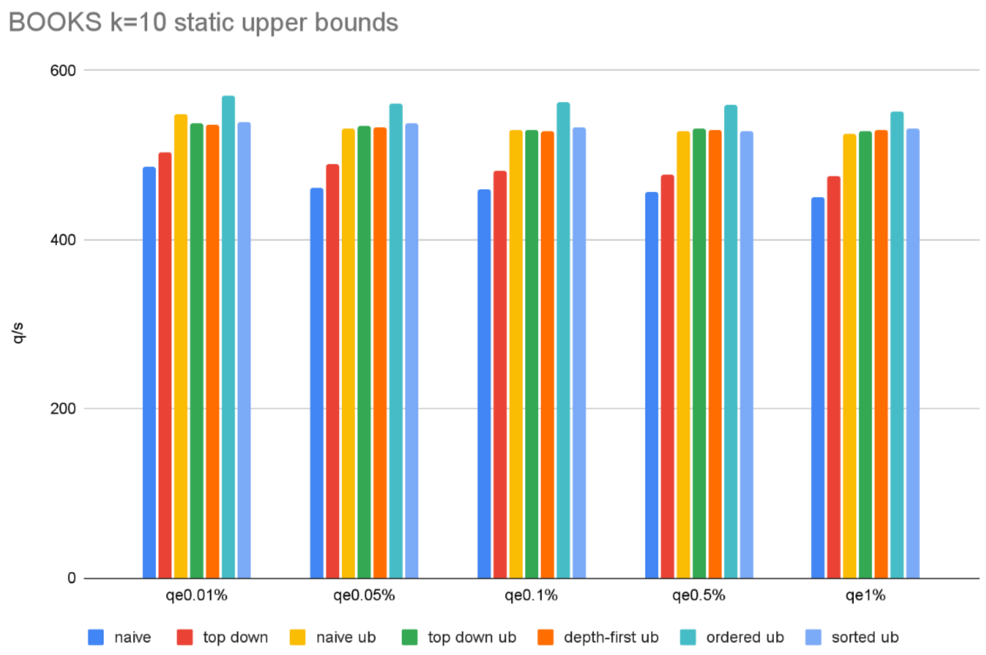


Figure 5.1: Throughput of methods on BOOKS dataset across query extents, k=10.

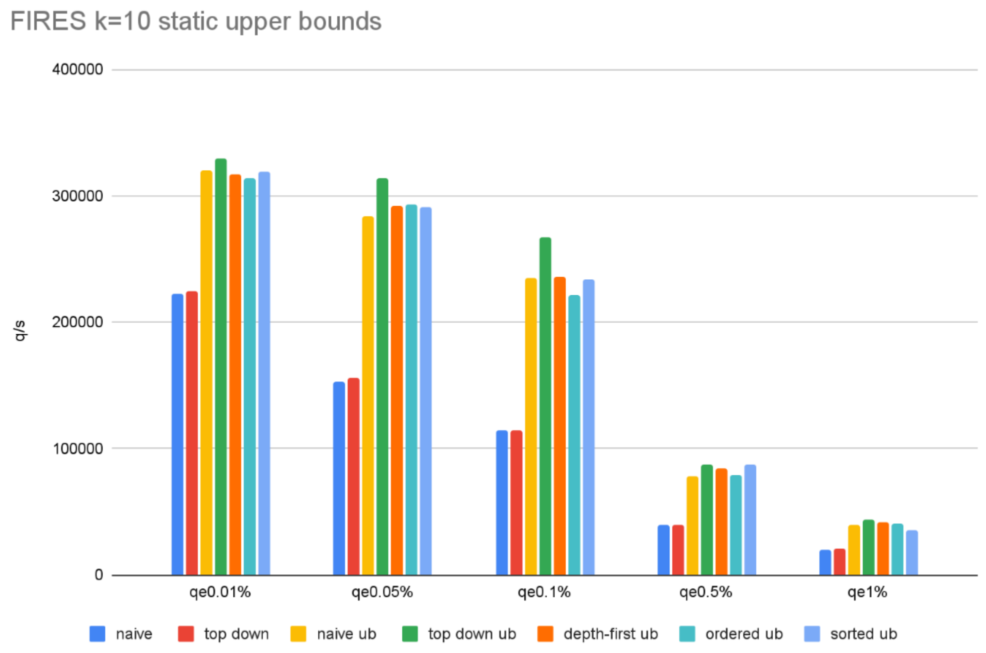


Figure 5.2: Throughput of methods on FIRES dataset across query extents, k=10.

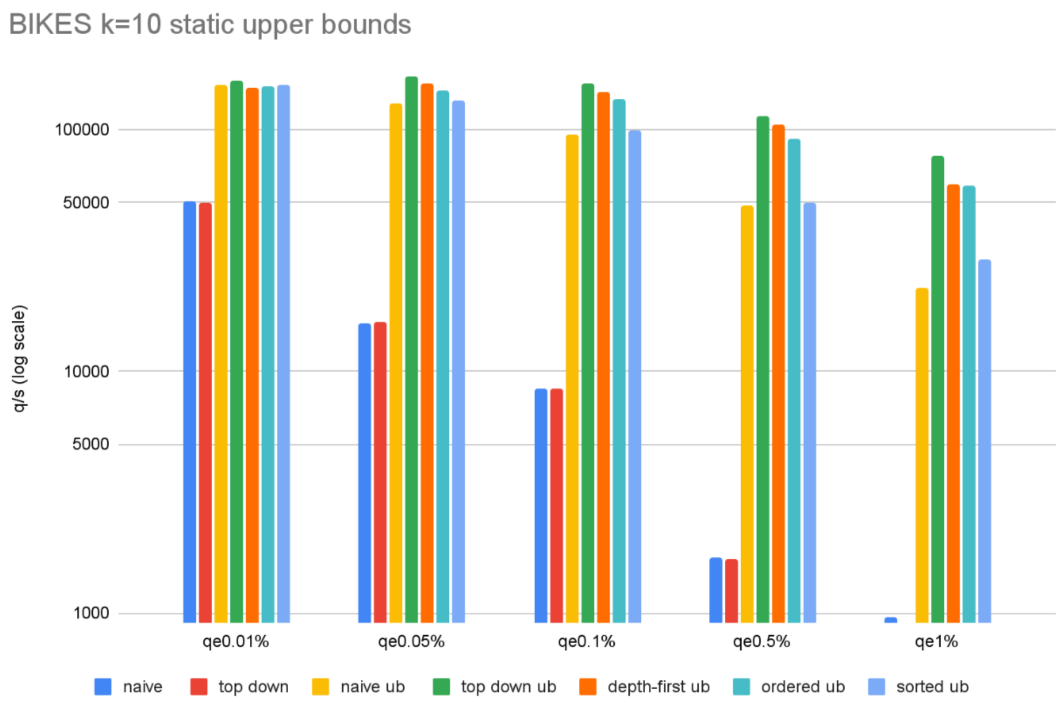


Figure 5.3: Throughput of methods on BIKES dataset across query extents, k=10

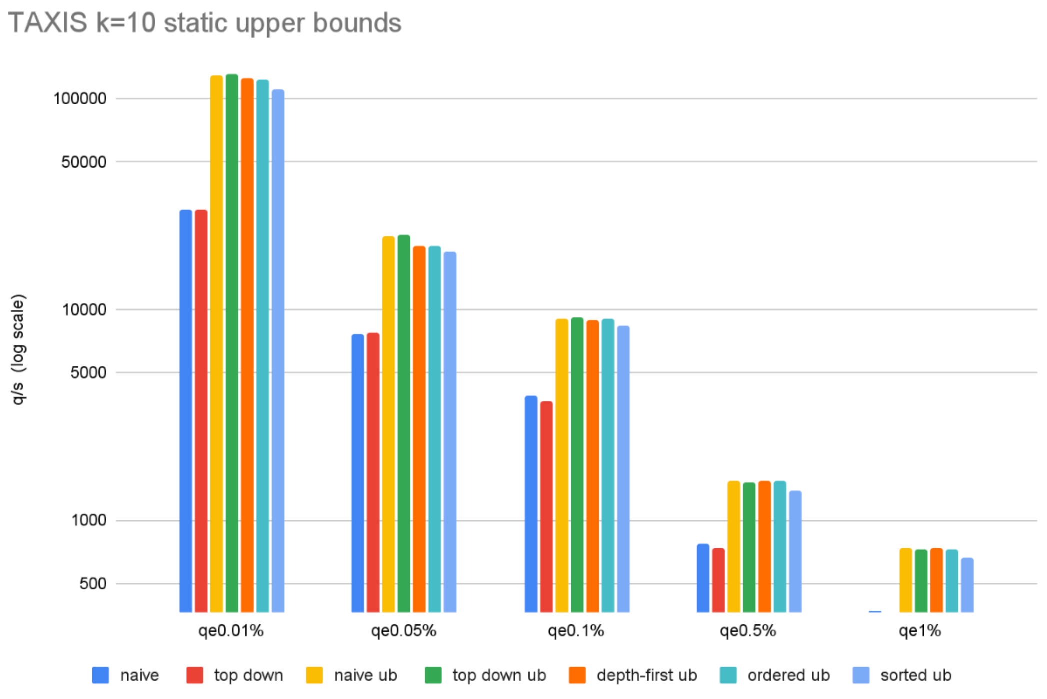


Figure 5.4: Throughput of methods on TAXIS dataset across query extents, k=10 (log scale).

5.4 Static vs Metadata Upper Bounds

Continuing our analysis, we compared the performance with the application of metadata upper bounds. Our observations indicate that the throughput consistently remains higher for all query extents when using metadata upper bounds compared to static ones. Still the dataset of “BOOKS” does not show any significant advantage in its performance. Notably, the ’top-down’ method, when coupled with metadata upper bounds, emerges as one of the best performing methods. Here, we depict only the methods when Upper Bounds are applied.

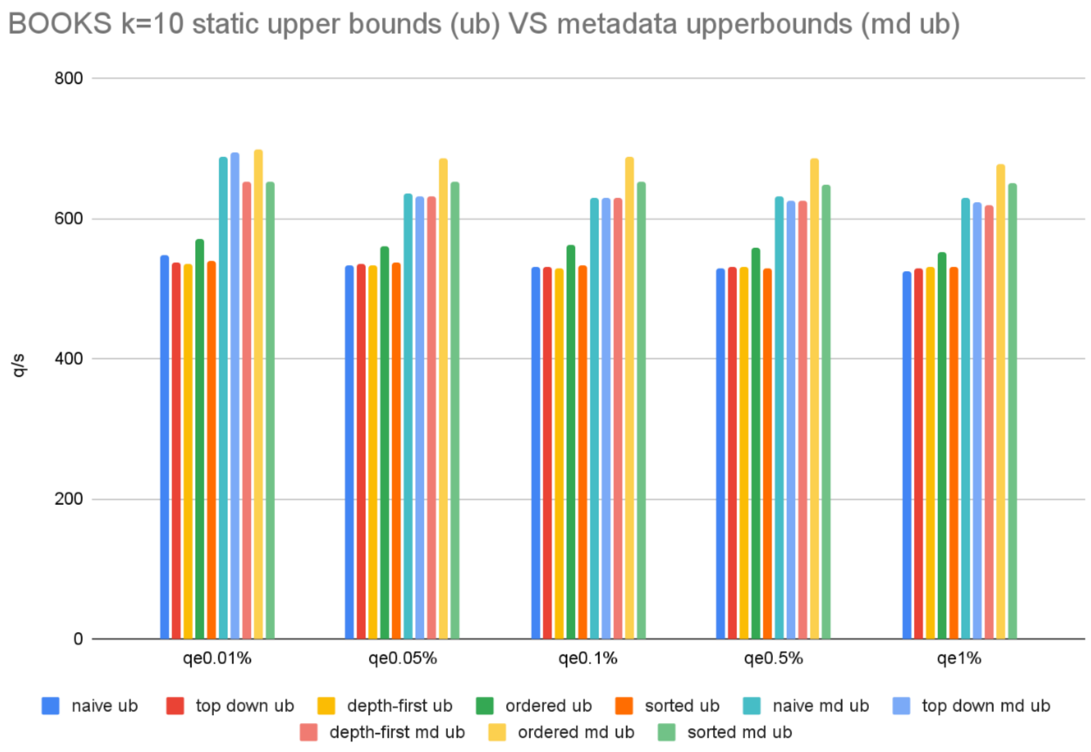


Figure 5.5: Throughput of methods with static and metadata Upper Bounds on BOOKS dataset across query extents, k=10.

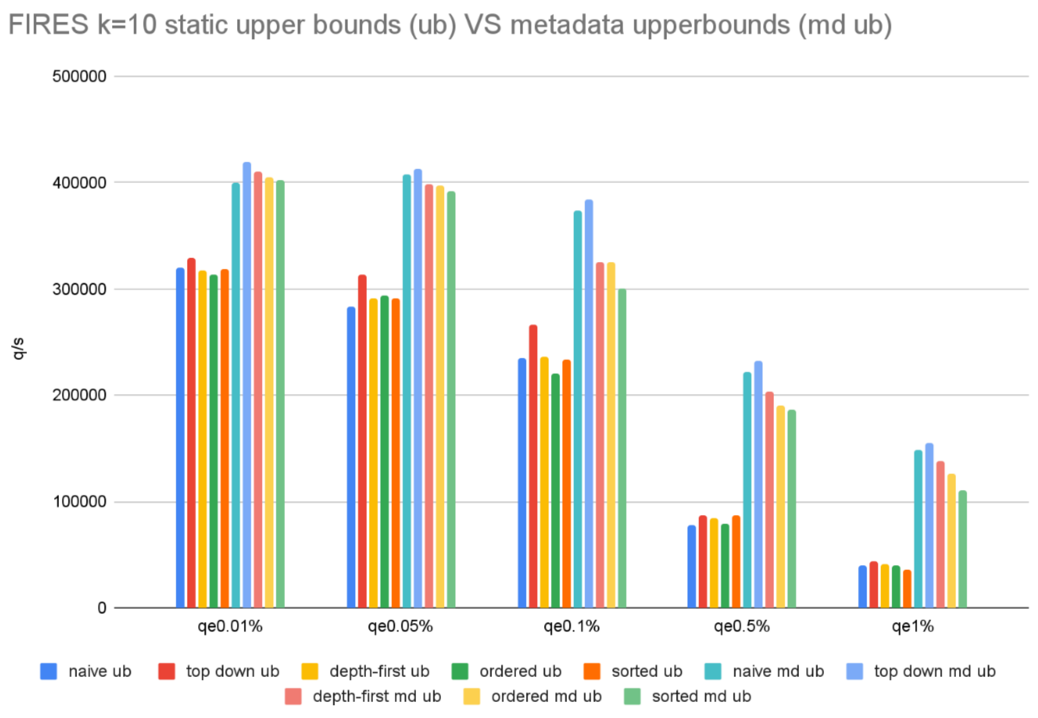


Figure 5.6: Throughput of methods with static and metadata Upper Bounds on FIRES dataset across query extents, k=10.

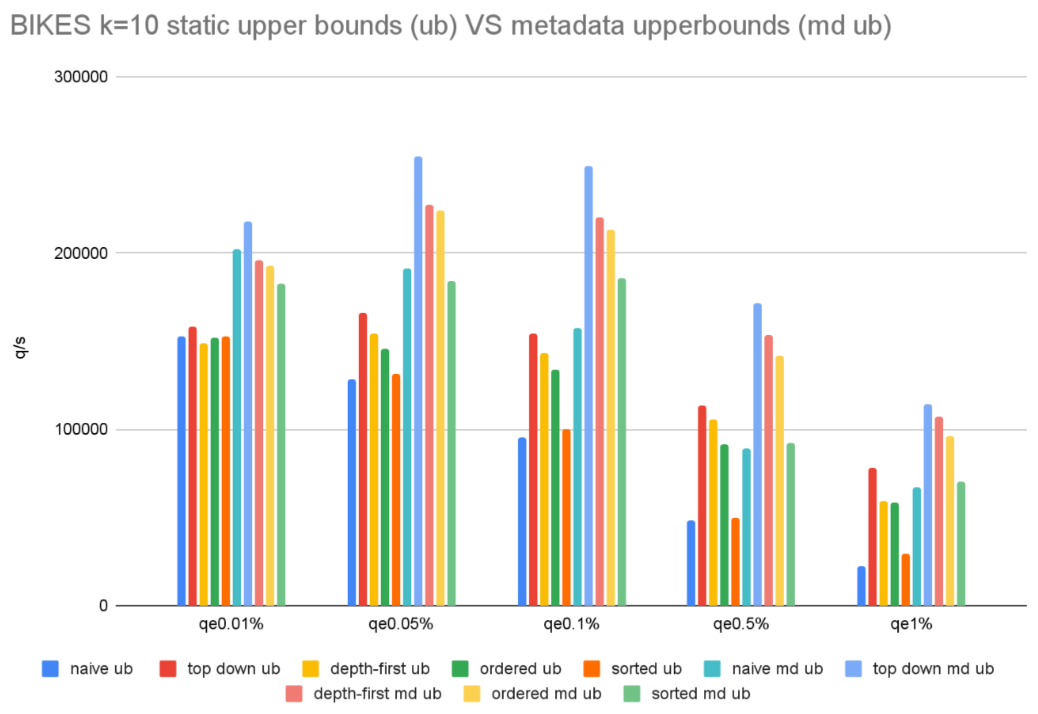


Figure 5.7: Throughput of methods with static and metadata Upper Bounds on BIKES dataset across query extents, k=10

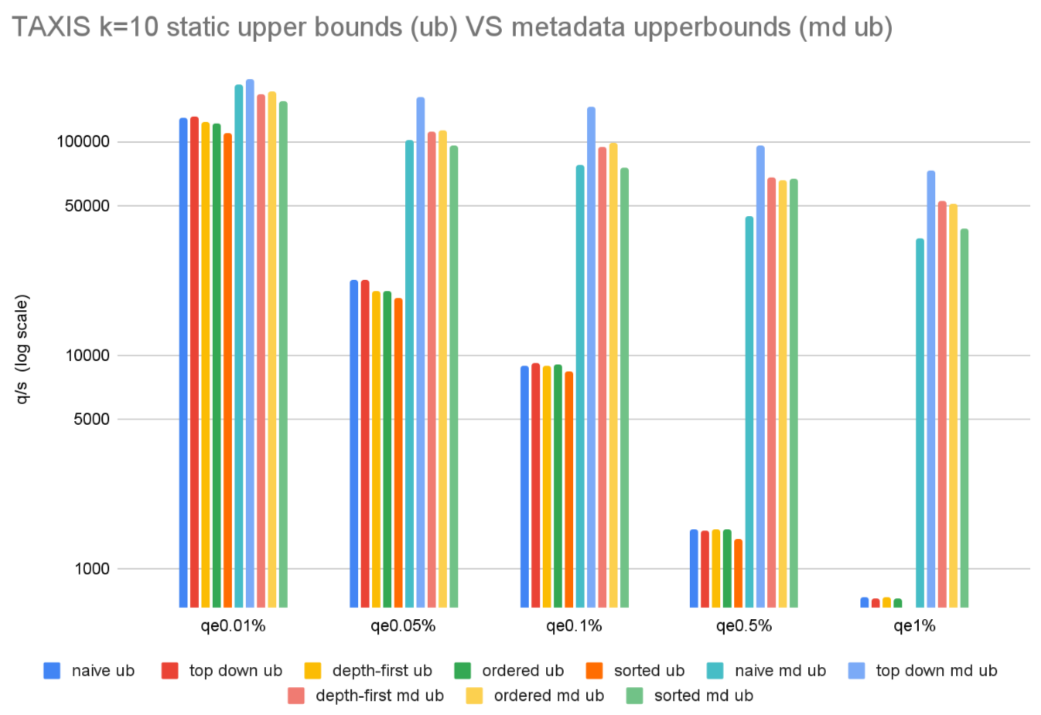


Figure 5.8: Throughput of methods with static and metadata Upper Bounds on TAXIS dataset across query extents, k=10 (log scale).

5.5 Parameter k

In the second stage of our experiments, we systematically varied the parameter k and repeated the aforementioned tests for each specified value: 5, 10, 50 and 100. This iterative approach enabled us to observe and analyze the performance characteristics of our methods when tasked with reporting larger sets of top k results. By adjusting k, we aimed to uncover how the scale of the result set influences system throughput across different datasets.

We present outcomes of the top-down method with and without upper bounds for different values of k, while keeping the query extent fixed at 0,1% of the initial domains. Notably, we exclude results from the “BOOKS” dataset, as it consistently did not exhibit any remarkable variations. The following figures detail the observed performance across the “TAXIS”, “BIKES”, and “FIRES” datasets.

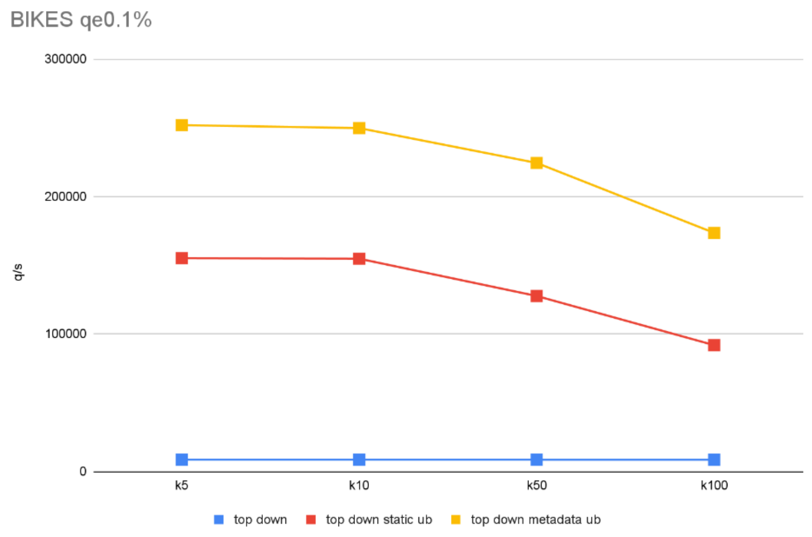


Figure 5.9: Scaling of Top-Down method (static & metadata Upper Bounds employed) on BIKES dataset for different values of k with query extent 0,1% of the domain.

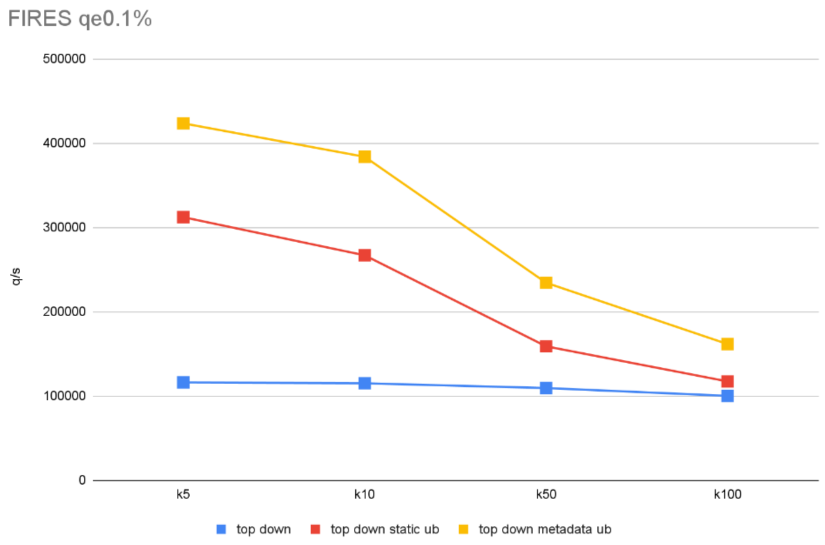


Figure 5.10: Scaling of Top-Down method (static & metadata Upper Bounds employed) on FIRES dataset for different values of k with query extent 0,1% of the domain.

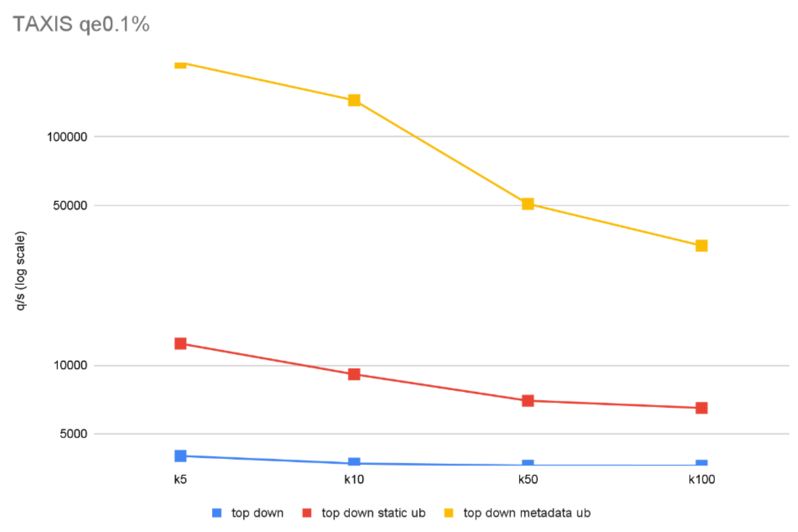


Figure 5.11: Scaling of Top-Down method (static & metadata Upper Bounds employed) on TAXIS dataset for different values of k with query extent 0,1% of the domain (log scale).

As we see, the throughput declines as we apply larger values of k for the three datasets, with 'BIKES' showing slightly better performance compared to 'TAXIS' and 'FIRES.' This pattern is consistent across the different query extents we tested. As well as for the methods that employ upper bounds.

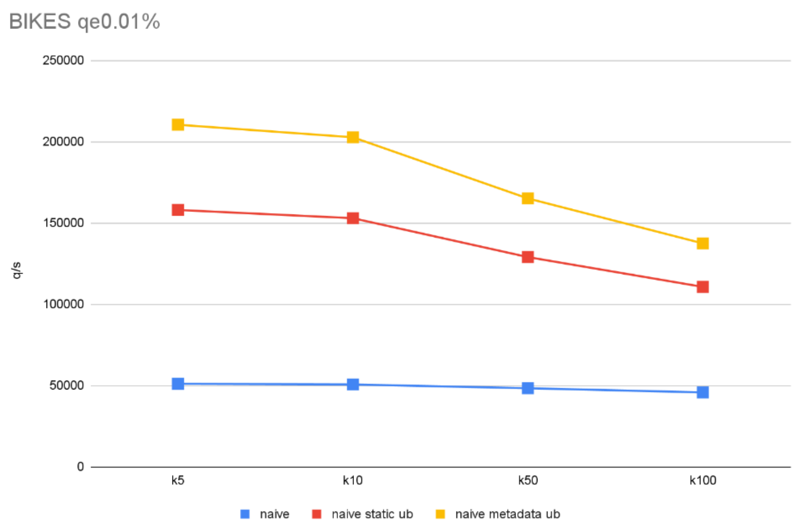


Figure 5.12: Scaling of Naive method (static & metadata Upper Bounds employed) on BIKES dataset for different values of k with query extent 0,01% of the domain (log scale).

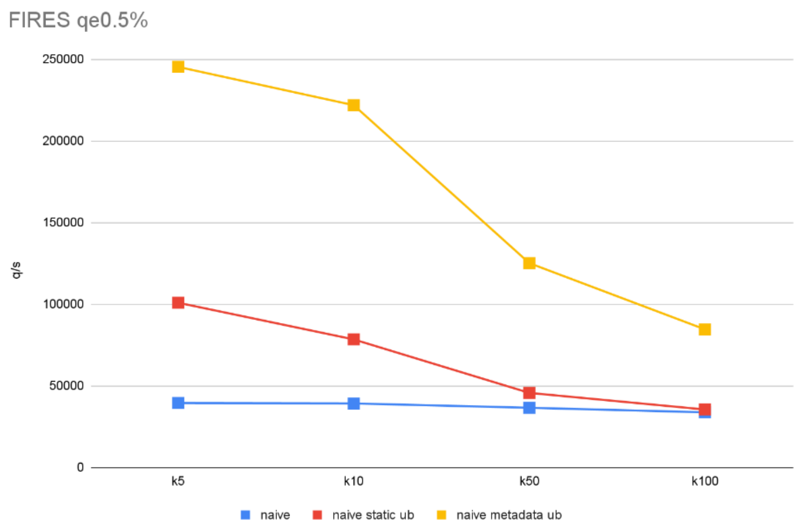


Figure 5.13: Scaling of Naive method (static & metadata Upper Bounds employed) on FIRES dataset for different values of k with query extent 0,5% of the domain (log scale).

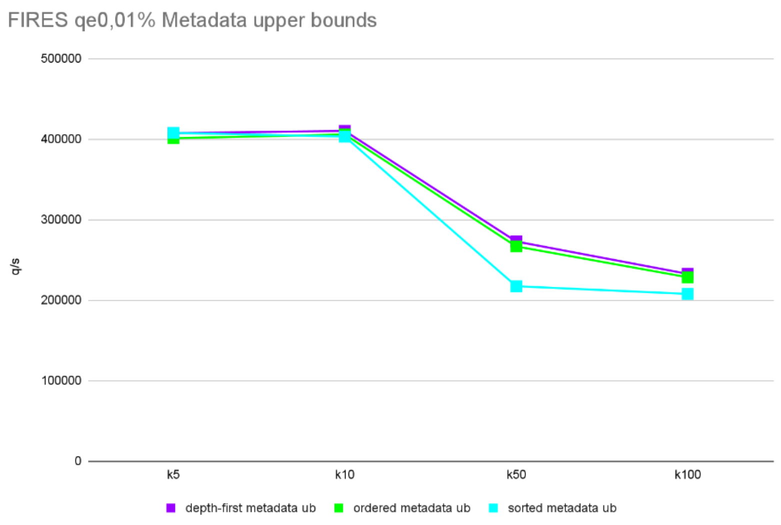


Figure 5.14: Scaling of Depth-First, Ordered & Sorted traversals on FIRES dataset for different values of k with query extent 0,01% of the domain.

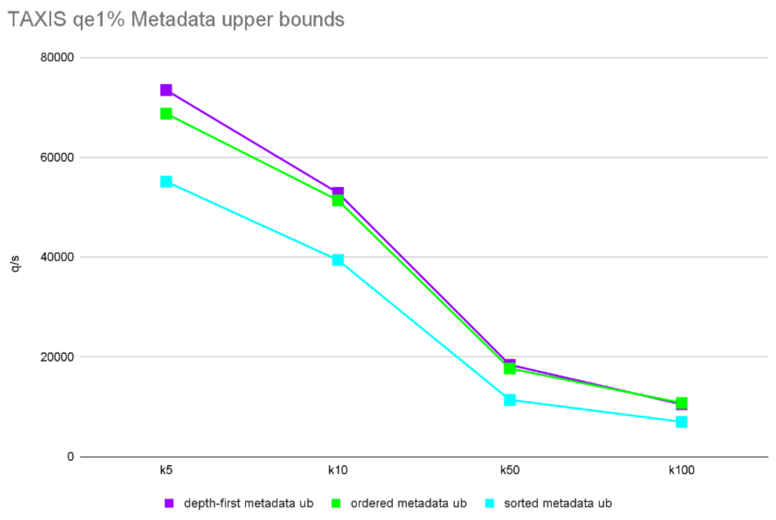


Figure 5.15: Scaling of Depth-First, Ordered & Sorted traversals on TAXIS dataset for different values of k with query extent 1% of the domain.

Summing up this section, our experimental evaluation highlights several key insights into the performance of top k query processing with index of HINTm under varying conditions. Firstly, larger query extents are associated with reduced throughput. However, the application of static upper bounds significantly accelerates the overall process, and extending these bounds to their metadata versions yields even better results. In terms of traversal methods, simpler approaches tend to offer superior performance. Specifically, the Top-Down method, which prioritizes traversing partitions with potentially larger intervals first, combined with metadata upper bounds, demonstrates the best performance among the tested methods.

Furthermore, the parameter k, representing the size of the top-k set, notably impacts implementation performance. Larger k values result in more frequent updates to the final set, thereby affecting throughput. Last but not least, the nature of the dataset plays a crucial role in system scalability. Datasets with larger intervals, such as 'BOOKS,' show different performance characteristics and appear less affected by the optimizations that benefit datasets with smaller intervals.

CHAPTER 6

CONCLUSIONS

6.1 Summary

We enhanced the HINTm index to efficiently handle ranking queries. Subsequently, we integrated a pruning technique utilizing Upper bounds, allowing the algorithm to bypass unnecessary partitions during the execution of top k queries. We further refined these Upper bounds to their metadata versions. In addition, we introduced novel methods that traverse the index in various ways, equipping these methods with the Upper bounds. Extensive experiments were conducted on four datasets to measure overall throughput. Initially, we assessed the system's performance across different query extents and then across varying k values. The results indicated that larger query extents and higher k values tend to reduce throughput. However, the application of Upper bounds, particularly the tighter metadata version, significantly improved the system's scalability, especially for datasets with relatively short average interval durations. Finally, among the developed methods, simpler approaches demonstrated slightly better performance, provided they were complemented by the metadata Upper bounds which has been proven to play the primary role in the overall performance.

6.2 Future Work

For future research, our initial objective is to explore alternative ranking functions beyond the absolute overlap. Subsequently, we will assess whether reporting results below a specified threshold score influences performance in comparison to reporting the top k results. Additionally, we intend to establish also lower bounds for each partition of the index. Lastly, we intend to study the performance of HINT on top k temporal joins.

References

[1] Michael H. Böhlen, Anton Dignös, Johann Gamper, and Christian S. Jensen. Temporal Data Management - An Overview. In eBISS 2017.

[2] Pierangela Samarati and Latanya Sweeney. Generalizing Data to Provide Anonymity when Disclosing Information. In ACM PODS 1998.

[3] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. Computational geometry: algorithms and applications, 3rd Edition. Springer 2008.

[4] Ahmed Awad, Riccardo Tommasini, Samuele Langhi, Mahmoud Kamel, Emanuele Della Valle, and Sherif Sakr. D2 IA: User-defined interval analytics on distributed streams. Information Systems 104 2022.

[5] George Christodoulou, Panagiotis Bouros, Nikos Mamoulis. HINT: A Hierarchical Index for Intervals in Main Memory. SIGMOD 2022.

[6] George Christodoulou, Panagiotis Bouros, Nikos Mamoulis. HINT: a hierarchical interval index for Allen relationships. VLDB 2023.

[7] Ronald Fagin, Amnon Lotem, Moni Naor: Optimal Aggregation Algorithms for Middleware. PODS 2001.

[8] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, David W. Cheung : Efficient top-k aggregation of ranked inputs. ACM Trans. Database Syst. 2007.

[9] Herbert Edelsbrunner. Dynamic Rectangle Intersection Searching. Technical Report 47. Institute for Information Processing, Technical University of Graz, Austria 1980.

[10] J. Dittrich and B. Seeger, “Data redundancy and duplicate detection in spatial join processing,” in IEEE ICDE 2000.

[11] Andreas Behrend, Anton Dignös, Johann Gamper, Philip Schmiegelt, Hannes Voigt, Matthias Rottmann, and Karsten Kahl. Period Index : A Learned 2D Hash Index for Range and Duration Queries. In SSTD 2019.

[12] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter M. Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index: a unified data structure for processing queries on temporal data in SAP HANA. In ACM SIGMOD 2013.

[13] R. Elmasri, G. T. J. Wuu, and Y. Kim, “The time index: An access structure for temporal data,” in 16th International Conference on Very Large Data Bases. Brisbane, Queensland, Australia, Proceedings, D. McLeod, R. Sacks-Davis, and H. Schek, Eds. Morgan Kaufmann, 1990.

[14] George Christodoulou. Interval Data Management in Main Memory. Ph.D. dissertation, Department of Computer Science and Engineering. University of Ioannina, Greece 2023.

[15] Richard T. Snodgrass and Ilsoo Ahn. 1986. Temporal Databases. Computer 19, 9 1986.