Interval Data Management in Main Memory

A Dissertation

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

by

George Christodoulou

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina
School of Engineering
Ioannina 2023

Dedication

To my beloved family

Acknowledgements

First and foremost, I would like to express my graditude to my advisor, Nikos Mamoulis, for the opportunity, the support, the guidance, and the patience he offered me through all the time we have worked together. I would like to thank my colleagues with whom we shared the lab through these years with our ups and downs. I am deeply grateful to my family for their unconditional support on any level that a human being can imagine. Their encouragement and faith in me were eternal through all these years. I am also deeply grateful to Eirini who supported and believed in me even when it defied common sense. Last but definitely not least, I would like to warmly thank Panos Bouros for all the knowledge and advice he gave me through the years. Panos shared his great experience in database systems and made me change my perspective about my work.

George Christodoulou

August 2023, Ioannina

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

George Christodoulou, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2023.
Interval Data Management in Main Memory.
Advisor: Nikos Mamoulis, Professor.

The management of intervals has been an active research area since databases were invented. A popular direction of research is the indexing and retrieval of intervals, finding a wide range of applications. Emerging and widely used systems are built dependent on temporal and uncertain data. Many algorithms and indices have been proposed, concentrated on a variety of queries. Most algorithms are either suboptimal in space consumption or perform well only for specific query types. We need novel and efficient in-memory indices for intervals, which can evaluate queries with high performance.

In statistical and probabilistic databases [3], uncertain values are often approximated by confidence intervals. Real-world examples of uncertain values include temperature values obtained from IoT devices or time-series. For such cases, it would be more appropriate to record an observation using an interval range rather than a single value. In data anonymization [4] attributes can be generalized to intervals. Stored values can be replaced with semantically consistent but less precise alternatives in the form of intervals. In this way, information from a private table, like the identity of any individual to whom the released data refer cannot be recognized. In XML data indexing techniques [5], the scope of an XML element can be modeled as an interval defined by the positions of the starting and closing tag of the element.

Intervals are representations of value ranges. Quite often, these ranges represent periods of time described as a tuple $[start, end]$. In a temporal database, an interval-based data model can timestamp each tuple or attribute value with a validity time interval. Along with valid time, an interval-based model can timestamp transaction

time, which captures when a tuple is inserted and deleted from the database. We index intervals in data structures so that we can efficiently evaluate different types of queries. There are several query types over intervals, with the differentiation lying on the specifications that shape the resulting set of intervals, or the context in which we model data with an interval representation.

The topic of this dissertation is to study the problem of indexing and querying a large collection of records, based on an interval attribute that characterizes each object. We focus on the different aspects of temporal databases, as they form the most significant application of interval data. The collection can be known before indexing or evolve over time, which is common in temporal databases or streaming data. The challenge is to find solutions which, can take advantage of modern hardware such as large main memories, can handle traditional and on demand indexing of intervals, and provide high performance for a wide variety of query types and predicates. In this thesis, we study numerous problems and different scenarios which come down to indexing and querying interval data.

In the first part, we propose HINT, a novel and efficient in-memory index for large known collections of intervals, with a focus on range queries, which are a basic component of many search and analysis tasks. Our index is suitable for valid-time indexing in the context of temporal databases. HINT applies a hierarchical partitioning approach, which assigns each interval to at most two partitions per level and has controlled space requirements. We reduce the information stored at each partition to the absolutely necessary by dividing the intervals in groups, based on whether they begin inside or before the partition boundaries. In addition, our index includes storage optimization techniques for the effective handling of data sparsity and skewness.

The second problem we study, is a more general version of HINT, so that with the best trade-off in information storage, it will be able to handle queries with different predicates. Intervals may satisfy more sophisticated relations than intersections, which are based on Allen's relationships [6] (e.g., find all intervals that are *covered by* the query interval). The principles of HINT are useful for the retrieval of data intervals based on Allen's relationships, because the hierarchical partitioning applies independently of the query type. We show how HINT can be tuned depending on the data and can be efficiently used to process joins and queries based on Allen's relationships.

In the last part of this dissertation, we study the problem of transaction-time indexing in the context of temporal databases, i.e., indexing versions of data in an evolving database. Given the fact that the main memories of modern commodity are large and cheap, we can afford to keep track of all versions of an evolving table in memory. This raises the question of how to index such a table effectively. We depart from the classic indexing approach, where both current (i.e., live) and past (i.e., dead) data versions are indexed in the same data structure, and propose LIT, a hybrid index, which decouples the management of the current and past states of the indexed column. LIT includes optimized indexing modules for dead and live records, which support efficient queries and updates, and gracefully combines them.

For the evaluation of our methods, we used multiple real and synthetic datasets with different characteristics so that we can safely conclude the robustness of our algorithms. The experiments showed that our algorithms are typically one order of magnitude faster than existing methods on static or evolving data collections and with multiple types of queries.

Εκτεταμενη Περιληψη

Γεώργιος Χριστοδούλου, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2023.
Διαχείριση Δεδομένων Εύρους στην Κύρια Μνήμη .
Επιβλέπων: Νίκος Μαμουλής, Καθηγητής.

Η διαχείριση των δεδομένων εύρους έχει αποτελέσει έναν ενεργό τομέα έρευνας από όταν εφευρέθηκαν οι βάσεις δεδομένων. Μια δημοφιλής κατεύθυνση έρευνας είναι η ευρετηρίαση και η ανάκτηση διαστημάτων, που βρίσκει ένα μεγάλο πεδίο εφαρμογών. Ένα μεγάλο σύνολο συστημάτων βασίζεται σε χρονικά και δεδομένα με ανακριβείς τιμές. Πολλοί αλγόριθμοι και ευρετήρια έχουν προταθεί, με στόχο την απάντηση διάφορων τύπων ερωτημάτων. Οι περισσότεροι αλγόριθμοι είναι είτε μη βέλτιστοι ως προς την χρήση χώρου είτε αποδίδουν καλά μόνο για συγκεκριμένους τύπους ερωτημάτων. Χρειαζόμαστε νέα και αποδοτικά ευρετήρια στην κύρια μνήμη για δεδομένα εύρους, τα οποία μπορούν να εκτελούν ερωτήματα με υψηλή απόδοση. Σε αυτήν τη διδακτορική διατριβή, στοχεύουμε στην έρευνα μεθόδων ευρετηρίασης, οι οποίες είναι ευέλικτες, έχουν χαμηλές απαιτήσεις χώρου και παρέχουν υψηλή απόδοση στην απάντηση ερωτημάτων.

Σε στατιστικές και πιθανοτικές βάσεις δεδομένων [3], δεδομένα με ανακριβείς τιμές συχνά προσεγγίζονται με εύρη (διαστήματα τιμών). Παραδείγματα πραγματικού κόσμου των δεδομένων με ανακριβείς τιμές περιλαμβάνουν μετρήσεις θερμοκρασίας που λαμβάνονται από συσκευές IoT ή καταγραφή χρονοσειρών. Για τέτοιες περιπτώσεις, θα ήταν πιο κατάλληλο να καταγράφεται μια παρατήρηση χρησιμοποιώντας ένα διάστημα τιμών αντί για μια μεμονωμένη τιμή ή μια ακολουθία παρόμοιων τιμών. Στην ανωνυμοποίηση δεδομένων [4], οι τιμές των χαρακτηριστικών των εγγραφών ενός πίνακα μπορούν να γενικευτούν σε εύρη. Οι αποθηκευμένες τιμές μπορούν να αντικατασταθούν με σημασιολογικά συνεπείς, αλλά λιγότερο ακριβείς εναλλακτικές αναπαραστάσεις υπό την μορφή εύρους. Με αυτόν τον τρόπο, πλη-

ροφορίες από πίνακες με ευαίσθητα δεδομένα, όπως η ταυτότητα οποιουδήποτε ατόμου στο οποίο αναφέρονται τα δεδομένα, δεν μπορούν να αναγνωριστούν. Στις τεχνικές ευρετηρίασης δεδομένων XML [5], το εύρος ενός στοιχείου XML μπορεί να μοντελοποιηθεί ως ένα εύρος που καθορίζεται από τις θέσεις της αρχικής και της κλείσιμος ετικέτας του στοιχείου.

Πολύ συχνά, αυτά τα εύρη αντιπροσωπεύουν χρονικά διαστήματα περιγραμμένα με την μορφή μιας πλειάδας $[\alpha, \beta]$. Σε μια χρονική βάση δεδομένων, ένα μοντέλο δεδομένων που βασίζεται σε εύρη μπορεί να αντιστοιχίσει την τιμή ενός γνωρίσματος με ένα αντίστοιχο χρονικό εύρος ισχύος. Εκτός από τον χρόνο ισχύος, ένα μοντέλο με βάση τα εύρη μπορεί να χαρακτηρίσει χρονικά τον χρόνο συναλλαγής, που απαθανατίζει τις στιγμές που μια πλειάδα εισάγεται και διαγράφεται από τη βάση δεδομένων.

Τα διαστήματα δεικτοδοτούνται από δομές δεδομένων για να αξιολογηθούν αποδοτικά διάφοροι τύποι ερωτημάτων. Υπάρχουν αρκετοί τύποι ερωτημάτων πάνω σε διαστήματα, με τη διάκριση τους να βρίσκεται στους περιορισμούς που διαμορφώνουν το σύνολο των διαστημάτων που προκύπτει ως επιθυμητό αποτέλεσμα, ή στο πλαίσιο με το οποίο μοντελοποιούμε τα δεδομένα μας.

Το θέμα αυτής της διατριβής είναι να μελετήσουμε το πρόβλημα της ευρετηρίασης και της αναζήτησης σε μια μεγάλη συλλογή δεδομένων, βασισμένη σε ένα εύρος που χαρακτηρίζει κάθε αντικείμενο. Η συλλογή μπορεί να είναι γνωστή πριν τη δημιουργία του ευρετηρίου ή να εξελιχθεί με τον χρόνο, κάτι το οποίο συνηθίζεται σε χρονικές βάσεις δεδομένων ή ροές δεδομένων. Η πρόκληση είναι να βρούμε λύσεις που μπορούν να εκμεταλλευτούν το σύγχρονο υλικό, όπως μεγάλες κύριες μνήμες, οι οποίες να μπορούν να χειριστούν την παραδοσιακή και την ευέλικτη ευρετηρίαση των ευρών και να παρέχουν υψηλή απόδοση για μια ευρεία ποικιλία τύπων ερωτημάτων.

Σε αυτήν τη διατριβή, μελετούμε πολλά προβλήματα και διαφορετικά σενάρια που συνδέονται με την ευρετηρίαση και την αναζήτηση δεδομένων διαστημάτων. Στο πρώτο μέρος, ασχολούμαστε με τη διαχείριση μη μεταβαλλόμενων (καλώς καθορισμένων) διαστημάτων τιμών, και προτείνουμε το HINT, ένα νέο και αποδοτικό ευρετήριο στην κύρια μνήμη, με εστίαση σε ερωτήματα εύρους, τα οποία αποτελούν βασικό στόχο της έρευνας. Το HINT εφαρμόζει μια ιεραρχική προσέγγιση ευρετηρίασης, που αναθέτει κάθε διάστημα σε το πολύ δύο διαμερίσματα ανά επίπεδο και έχει ελεγχόμενες απαιτήσεις χώρου. Μειώνουμε την πληροφορία που αποθηκεύουμε για τα δεδομένα στο ελάχιστο, χωρίζοντας τα δεδομένα σε ομάδες βάσει του αν η αρχή τους βρίσκεται εντός ή εκτός από τα όρια ενός διαμερίσματος. Επιπλέον, το ευρετήριο μας περιλαμβάνει τεχνικές βελτιστοποίησης αποθήκευσης για την αποτελεσματική διαχείριση αραιών και ασύμμετρα κατανεμημένων στον χώρο δεδομένων.

Το δεύτερο πρόβλημα που μελετούμε είναι μια γενικευμένη έκδοση του HINT, έτσι ώστε με την κατάλληλη ρύθμιση αποθήκευσης πληροφοριών, να μπορεί να χειριστεί ερωτήματα με διαφορετικούς περιορισμούς. Τα εύρη μπορεί να ικανοποιούν πιο εξεζητημένες σχέσεις, οι οποίες περιγράφονται στην βιβλιογραφία ως Άλγεβρα του Allen [6] (π.χ. εύρεση όλων των εγγραφών που *επικαλύπτονται από* το εύρος του ερωτήματος). Οι αρχές του HINT είναι χρήσιμες για την ανάκτηση δεδομένων εύρους βάσει των σχέσεων του Allen, καθώς η ιεραρχική διαίρεση εφαρμόζεται ανεξάρτητα από τον τύπο του ερωτήματος.

Στο τρίτο (και τελευταίο) μέρος αυτής της διατριβής, μελετούμε το πρόβλημα της ευρετηρίασης δεδομένων στο πλαίσιο χρόνου συναλλαγών, δηλαδή την ευρετηρίαση δεδομένων με πολλαπλές εκδόσεις σε μια συνεχώς εξελισσόμενη βάση δεδομένων. Δεδομένου ότι οι κύριες μνήμες των σύγχρονων υπολογιστών είναι μεγάλες και φθηνές, μπορούμε να επιτρέψουμε την αποθήκευση όλων των εκδόσεων ενός εξελισσόμενου πίνακα στη μνήμη. Αυτό θέτει το εξής ερώτημα: πώς δημιουργήσουμε αποτελεσματικά ένα ευρετήριο για έναν τέτοιο πίνακα; Σε αντίθεση με την κλασική προσέγγιση ευρετηρίασης, όπου τόσο οι τρέχουσες (δηλαδή ενεργές) όσο και οι παλιές (δηλαδή ανενεργές) εκδόσεις δεδομένων συνυπάρχουν στην ίδια δομή δεδομένων, προτείνουμε το LIT, ένα υβριδικό ευρετήριο, που ξεχωρίζει τη διαχείριση των ενεργών και ανενεργών καταστάσεων των δεδομένων. Το LIT περιλαμβάνει τεχνικές βελτιστοποίησης για την αποδοτική ευρετηρίαση, και υποστηρίζει αποτελεσματικά ερωτήματα αλλά και ενημερώσεις στις εκδόσεις των δεδομένων.

Για την συνολική αξιολόγηση όλων των μεθόδων μας, χρησιμοποιήσαμε πολλά πραγματικά και συνθετικά σύνολα δεδομένων με ποικίλα χαρακτηριστικά, ώστε να μπορούμε να συμπεράνουμε με ασφάλεια την αξιοπιστία των αλγορίθμων μας. Τα πειράματα δείχνουν πως οι αλγόριθμοί μας είναι, στην γενική περίπτωση, μια τάξη μεγέθους πιο γρήγοροι από τις υπάρχουσες μεθόδους σε συλλογές στατικών ή εξελισσόμενων δεδομένων και με διαφορετικούς τύπους ερωτημάτων.

Chapter 1

Introduction

A wide range of applications require managing large collections of intervals. In data anonymization [4], attribute values are often generalized to value ranges. XML data indexing techniques [5] encode label paths as intervals and evaluate path expressions using containment relationships between the intervals. Several computational geometry problems [7] (e.g., windowing) use interval search as a module. The internal states of window queries in Stream processors (e.g. Flink/Kafka) can be modeled and managed as intervals [8]. The most popular use of intervals is time representation in temporal databases [9, 10]; where each tuple has a *validity interval*, which captures the period of time that the tuple is *valid* in the modeled reality. Temporal data management has been studied extensively for at least four decades [9, 11, 12, 13, 10]. Temporal and multi-version data management re-gained interest recently [14, 2, 15, 16, 17, 18, 19, 20, 21, 22], due to the increase of cheap storage that makes it possible to track the versions of a database even in the main memory of a commodity machine.

Temporal databases are a fundamental application of interval data because they are specifically designed to manage and store time-varying data. Interval representations can be used to model time in two ways [23]: *valid time representations* and *transaction* *time representations*. The former captures the time during which a fact or event holds true in the reality of an application, while the latter records the time when data were recorded or modified in the database. In this dissertation, we study the indexing of interval data, with a primary focus on addressing the unique challenges that arise from different time modeling and different query types, and proposing novel solutions for effective query processing and data retrieval.

Valid time indexing is essential for applications where the validity of data items (e.g., records, attribute values) is limited to specific time intervals. (e.g., records, attribute values) is limited to specific time intervals. Among the most common and popular types of queries encountered in this domain are the range queries. These queries aim to identify intervals that overlap in any way with a specified (time) range, often employed in scenarios where historical trends, scheduling, or event detection are of high importance. The predicate of this query is denoted as `G-OVERLAPS`. Range queries are also known as pure timeslice/timerange queries in temporal databases [24]. Although range queries are fundamental and very popular, previous work has mainly focused on more expensive and complex queries, such as interval joins [25, 26, 27, 28, 10, 29] or temporal aggregation [30, 31, 32, 33, 34]. For efficient range queries over collections of intervals, classic data structures for managing intervals, like the interval tree [35], are typically used.

1.1 Interval Indexing

To tackle the challenges of interval indexing, in the first part of this dissertation, we introduce a new indexing method (HINT), a general-purpose index for intervals in a static domain and can handle valid time data. HINT efficiently answers range `G-OVERLAPS` queries, enabling fast retrieval of relevant intervals. Our approach leverages optimization techniques and data structures designed to handle the temporal nature of the data, enhancing query performance while keeping size efficiency. Subsequently, to validate the superiority of our method, we conduct an extensive set of experiments, comparing its performance against state-of-the-art techniques in the domain of interval data indexing.

Intervals can involve more intricate relationships than simple intersections, making it necessary to adopt advanced formalisms to capture these complex associations accurately. Allen's Algebra [6], introduced by James F. Allen in 1981, is a well-established and widely used framework for characterizing relationships between intervals. This model defines a set of 13 basic interval relations, also known as Allen's predicates, that describe the possible relationships between two intervals (e.g. "before", "meets"). Recognizing the significance of these sophisticated interval relationships, the second part of this dissertation focuses on extending and fine-tuning HINT to support queries with Allen's predicates. The objective is to efficiently answer queries that involve Allen's predicates, allowing for more complex interval data analysis. By leveraging the capabilities of Allen's Algebra, the indexing approach can handle queries that go beyond basic intersections, facilitating more advanced interval data retrieval and analytics.

Contribution In the first two chapters of this dissertation, focused on static interval data, we propose a novel and general-purpose Hierarchical index for INTervals (HINT), suitable for applications that manage large collections of intervals. HINT defines a hierarchical decomposition of the domain and assigns each interval in $\mathcal{S}$ to at most two partitions per level. If the domain is relatively small and discrete, our index can evaluate `G-OVERLAPS` queries, requiring no comparisons at all. For the general case where the domain is large and/or continuous, we propose a version of HINT, denoted by HINT^m, which limits the number of levels to $m + 1$ and greatly reduces the space requirements. HINT^m conducts comparisons only for the intervals in the first and last accessed partitions at the bottom levels of the index. Some of the unique and novel characteristics of our index include:

- The intervals in each partition are further divided into groups, based on whether they begin inside or before the partition. This division (1) cancels the need for detecting and eliminating duplicate query results, (2) reduces the data accesses to the absolutely necessary, and (3) minimizes the space needed for storing the objects into the partitions.

- As we theoretically prove, the expected number of HINT^m partitions for which comparisons are necessary is at most four. This guarantees fast retrieval times, independently of the query extent and position.

- The optimized version of our index stores the intervals in all partitions at each level sequentially and uses a dedicated array with just the ids of intervals there, as well as links between non-empty partitions at each level. These optimizations

Table 1.1: Comparison of interval indices

Method	**query cost**	**space**	**updates**
Interval tree [35]	medium	low	slow
Timeline index [13]	medium	medium	slow
1D-grid	medium	medium	fast
Period index [36]	medium	medium	fast
HINT/HINT^m (our work)	low	low	fast

facilitate sequential access to the query results at each level, while avoiding accessing unnecessary data.

- We propose a model for tuning the value of the parameter m for HINT^m. Furthermore, we include experiments which confirm the intuition behind our proposed model.

- Our experimental evaluation on real and synthetic datasets shows that our index is typically *one order of magnitude faster* than the competition. Table 1.1 qualitatively compares HINT to previous work.

- We show the necessary additional comparisons and accesses on HINT^m for each predicate in Allen's algebra. In addition, we show that a different version of HINT^m is directly suitable for processing queries using all Allen's predicates, while maintaining the excellent performance of HINT^m for `G-OVERLAPS` queries.

- We show that an index-based nested loops approach for `G-OVERLAPS` interval joins that uses HINT^m to index the inner join input outperforms the state-of-the-art join algorithm when the outer join input is relatively small.

1.2 Indexing Intervals for Transaction Time Temporal Databases

In transaction time temporal databases, the focus shifts from capturing the validity of data in the application domain to recording the time at which data were inserted or modified in the database. Unlike valid time, which represents when facts or events are true in the reality of the application, transaction time represents when these facts or events were captured or changed within the database system. Also, there is a necessity of considering both, the temporal aspect and the accompanying attributes during query processing. Each data entry is associated with a timestamp that reflects the exact moment when the data were inserted, updated, or deleted in the database. As new data arrive or existing data are modified, the corresponding timestamps are updated to reflect these changes. When a data entry is firstly inserted the only known timestamp is the starting point of its validity interval. Among the most common encountered query types within this field are referred to as *time-travel queries*. These queries are categorized into two main groups: *pure timeslice/timerange queries*, which are the same as previously mentioned, and additionally, *range timeslice/timerange queries*, that also include selection predicates on the non-temporal aspect of the data as well.

As an example, consider a database table T, storing information about employees of a company. The table has three attributes: ID, Name, and Salary. As the database evolves over time, there are changes in the table, where records are inserted or deleted, or attribute values of existing records are updated. Figure 1.1 shows some versions of T, where, at time t_0, T is initialized to include two records; at time t_1, a new record (with ID=3) is inserted to T; at time t_2, the Salary value of record 2 is updated; and at time t_3, record 1 is deleted and record 2 is updated. The evolution of T can be seen as a stream (time-sequence) of events, also shown in the figure (bottom-left). Insertions (deletions) are modeled by events of type *start* (*end*); each update (i.e., value changes) is modeled by a deletion immediately followed by an insertion. Finally, the figure (bottom-right) shows the *validity* intervals of the records and their values in the Salary attribute, as flat line segments. The current time is denoted by t_{now}. We first focus on indexing for *pure* time travel queries, where the objective is to retrieve the record versions that were valid at a given *timepoint* or *timerange* in the past. In our running example (Figure 1.1), such a pure timepoint query q_0 is "find all records in T, which were valid at time t_{q0}" and the answer records are (1, Smith, 50K) and (2, Black, 30K). Then, we study how our indexing scheme can be extended to temporally index T with respect to a specific attribute $T.A$, for *range* time travel queries, that retrieve record versions r in T which were valid at a given timepoint/timerange and their $r.A$ satisfies a range query predicate. Such a range-timepoint query q_1 is: "find all records in T, which were valid at time t_{q1} and have Salary at most 32K." Query q_1 is geometrically represented by the vertical line segment starting at time t_{q1} and retrieves the records, corresponding to the line segments intersected by vertical segment starting at t_{q1}, i.e., record (2, Black, 30K). Another example is range-timerange query q_2: "find all records in T, which were valid

ID	Name	Salary
1	Smith	50K
2	Black	30K

after t_0

ID	Name	Salary
1	Smith	50K
2	Black	30K
3	James	40K

after t_1

ID	Name	Salary
1	Smith	50K
2	Black	35K
3	James	40K

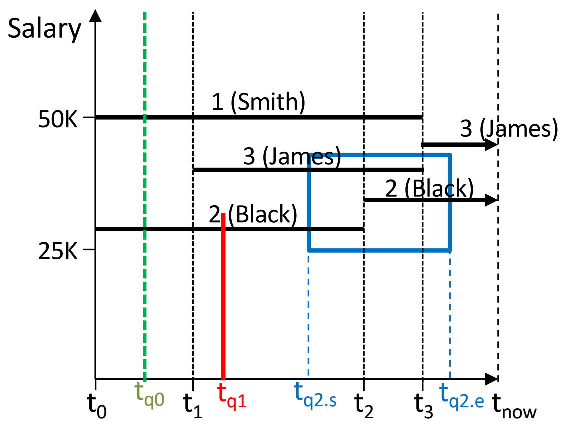
after t_2

ID	Name	Salary
2	Black	35K
3	James	45K

after t_3

EventID	Time	r.ID	r.Salary	Event
0	t_0	1	50K	start
1	t_0	2	30K	start
2	t_1	3	40K	start
3	t_2	2	30K	end
4	t_2	2	35K	start
5	t_3	1	50K	end
6	t_3	3	40K	end
7	t_3	3	45K	start

Events sequence



Geometric representation

Figure 1.1: Example of a time-evolving table

anytime between $t_{q2.s}$ and $t_{q2.e}$ and have Salary between 25K and 43K," modeled by the rectangle in Figure 1.1. Again, the query results are the segments that intersect the rectangle, namely (2, Black, 30K) valid in $[t_0, t_2)$, (2, Black, 35K) valid in $[t_2, t_{now})$, and (3, James, 40K) valid in $[t_1, t_3)$. Note that it is important to find the records *and* their validity intervals in order to be able to distinguish between results corresponding to different versions of the same record/entity (e.g., Black in the results of q_2).

To address transaction time indexing, in the third (and last) part of this dissertation, we propose a novel indexing solution tailored to fit the specifications of the transaction time aspect of temporal databases. Our approach uses advanced data structures and indexing techniques that facilitate fast retrieval while efficiently considering both temporal information and the associated attributes. By combining temporal and attribute-based querying, our index forms a complete solution, useful in various application domains.

Contribution We aim at the efficient support of updates in a continuously evolving database, and target a much better performance in queries compared to the state-of-the-art access methods for time-evolving data.

- We propose a LIT, a *hybrid* index, which indexes *live* records (i.e., those which valid at t_{now}), like (2, Black, 35K), by a different data structure compared to *dead* records (i.e., those not currently valid), like (2, Black, 30K). Specifically, LIT includes a LiveIndex for the live records; LiveIndex only needs to index the begin time of the validity of each live record. For dead records we use a DeadIndex, which includes their validity intervals with both starting and ending timepoints. When a temporal record is created, it is added to LiveIndex; when the record dies (i.e., deleted from the temporal table T, or updated), it is deleted from LiveIndex and added to the DeadIndex. Given these operations, LiveIndex supports fast *temporal appends* (i.e., add a new live record at the "temporal" end of the index) and deletions, whereas DeadIndex needs only to support insertions (anywhere in the time domain up to t_{now}), but no deletions (since past data versions are never deleted from a temporal DB). Both LiveIndex and DeadIndex gracefully adapt to the ever-evolving time domain.

- We implement, tune and test the best implementations of LiveIndex and DeadIndex and compare LIT with in-memory versions of the state-of-the-art temporal and multi-version indices [37, 13] on mixed workloads of queries and version updates, showing that LIT is orders of magnitude faster.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we describe the necessary background and useful definitions for our work. In Chapter 3, we review related work and present in detail the characteristics and weaknesses of existing methods.

In Chapter 4, we present HINT and its generalized HINT^m version, and analyze their complexity. We focus primarily on the `G-OVERLAPS` relationship and optimizations that boost the performance of HINT^m. Then we focus on executing interval joins with HINT. Last, we present our experimental analysis on real and synthetic data against the state-of-the-art.

In Chapter 5 we discuss necessary changes to HINT^m for efficiently evaluating selection queries under the Allen's algebra relationships, and evaluate our method experimentally.

In Chapter 6, we present LIT, our proposal for pure time-travel queries on transaction-time databases and its extension so that it can index an attribute A of the records besides their temporal validity intervals, in order to support range time-travel queries. We conclude this chapter with a discussion about the integration of our main-memory LIT in a DMBS that should support persistence and fault-tolerance (recovery) and our experimental analysis. In conclusion, Chapter 7 summarizes the contributions of this dissertation and provides a discussion about future work.

Chapter 2

Background and Definitions

In this chapter, we present the background and definitions that form the basis of our research. We will take a closer look at the context that has shaped our study's foundation. By clarifying key terminologies and concepts, we aim to provide a better understanding of the ideas we are going to explore through the rest of this dissertation. These insights will serve as a solid platform for our analysis on interval data management.

In mathematics, an interval is a fundamental concept used to describe a continuous range of numbers. There are various types of intervals, each with its own formal definition. Here are the basic interval formal definitions:

Closed Interval 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.

Open Interval An open interval is a set of real numbers that includes all the numbers between its endpoints but excludes the endpoints themselves. It is denoted by (a, b), where "a" and "b" are the two endpoints, and all numbers between a and b (excluding a and b) belong to the interval.

Half-Open or Half-Closed Interval A half-open interval is a set of real numbers that includes one endpoint and excludes the other. There are two types of half-open intervals:

a. Right Half-Open Interval: Denoted by $[a, b)$, it includes a and all numbers between a and b (excluding b).

b. Left Half-Open Interval: Denoted by $(a, b]$, it includes b and all numbers between a and b (excluding a).

Infinite Interval An interval can also be infinite when one or both of its endpoints are not defined (positive or negative infinity). Denoted by $(-\infty, \infty)$, it includes all real numbers.

In our setting, without loss of generality, we assume that the validity interval of a record is closed at both ends. Intervals are indexed by data structures in order to efficiently evaluate different types of queries. There are several query types over intervals, so different data structures may be needed for their efficient evaluation. We model our problem as indexing a large collection $\mathcal{S}$ of objects (or records), based on an interval attribute that characterizes each object. Hence, we model each object $s \in \mathcal{S}$ as a triple $\langle s.id, s.st, s.end \rangle$, where $s.id$ is the object's identifier (which can be used to access any other attribute of the object), and $[s.st, s.end]$ is the interval associated to s. These query types can be described by the following definitions:

Stabbing queries (or pure timeslice queries in the context of temporal databases) ask for the intervals in the database (or the objects associated with them), which include a query value x. For example, interval $[6, 9]$ is a result for the query value $x = 7$. (Note that the predicate can be considered as G-OVERLAPS)

Interval range queries (or pure timerange queries in the context of temporal databases) retrieve intervals in a collection of intervals, which overlap (i.e., have at least one common value) with a given query interval x. For example, interval $[6, 9]$ is a result for an interval range query with $x = [3, 7]$. (Note that the predicate can be considered as G-OVERLAPS)

Relationship G-OVERLAPS A G-OVERLAPS selection retrieves all intervals intersecting query q in any way.

Relationship EQUALS. An EQUALS selection retrieves all input intervals identical to query q, i.e., with $q.end = s.end$ and $q.st = s.st$.

Relationship STARTS. According to Allen's algebra, a STARTS selection query reports all intervals that *start* where q does, i.e., with $q.st = s.st$, but outlive its *end*, i.e., with $q.end < s.end$.

Relationship STARTED_BY. As an inverse to STARTS, a STARTED_BY selection retrieves all intervals that again *start* at $q.st$ but *end* before $q.end$.

Relationship `FINISHES`. This selection query returns all intervals that *end* exactly where query q does, i.e., with $q.end = s.end$, but *start* before q, i.e., with $q.st > s.st$.

Relationship `FINISHED_BY`. A `FINISHED_BY` selection inverses the second condition of `FINISHES`, retrieving intervals with $q.end = s.end$ and $q.st < s.st$.

Relationship `MEETS`. This selection query returns all intervals that *start* at $q.end$.

Relationship `MET_BY`. This selection query returns all intervals that *end* at $q.st$.

Relationship `OVERLAPS`. An `OVERLAPS` selection retrieves all non-disjoint intervals to query q, which start *after* $q.st$ and end *after* $q.end$.

Relationship `OVERLAPPED_BY`. As inverse to `OVERLAPS`, the `OVERLAPPED_BY` selection retrieves all non-disjoint intervals to q that start *before* $q.st$ and end *before* $q.end$.

Relationship `CONTAINS`. This selection query returns all intervals, fully *contained* inside the query interval q, i.e., with $q.st < s.st \wedge q.end > s.end$.

Relationship `CONTAINED_BY`. This selection retrieves all intervals that fully *contain* q, i.e., with $q.st > s.st \wedge q.end < s.end$.

Relationship `BEFORE`. A `BEFORE` selection retrieves all intervals that start *after* q.

Relationship `AFTER`. An `AFTER` selection retrieves all intervals that end *before* q.

In a data modeling context, there are distinct time dimensions that can be considered to represent temporal aspects of the modeled reality and database records. These time dimensions are "valid time" and "transaction time", and they serve different purposes in capturing time-varying states and maintaining data in a database.

Valid Time (or Application Time) refers to the time during which a fact is true in the modeled reality. It represents the temporal validity of the information being stored in the database. For example, if we consider the fact "George was hired from September 1, 2010, to March 30, 2012", the valid time is the period between these dates when George's hiring status is considered valid. Furthermore, valid time is supplied by the application and exists independently of whether the fact is recorded in a database or not. Also, facts in the data model have valid time by definition. Valid time can be bounded (limited within specific time intervals) or unbounded (continuing indefinitely).

Transaction Time (or System Time) represents the time when a fact is present or current in the database as stored data. It captures the temporal aspect of the database's changing state over time. In the same example, “George was hired from September 1, 2010, to March 30, 2012”, although the transaction time is the period between when this hiring information was inserted into the database (e.g., September 5, 2010) and when it was deleted (e.g., April 2, 2012). The transaction time aspect is supplied automatically by the DBMS. It has a duration from the insertion of a fact to its deletion, allowing for multiple insertions and deletions of the same fact. Deletions in transaction time are logical, meaning the fact remains in the database but is no longer part of the database’s current state. When a new fact is inserted into the database, it becomes part of the current state of the database, and it is associated with a specific transaction time that marks the moment of insertion. This fact is now a version of the data, representing the state of the database at that particular transaction time.

The activities in transaction time databases generate multiple versions of a fact or record as it evolves over time. This organic versioning occurs due to the nature of how data is inserted, updated, and deleted in the database, capturing the changes in the database’s state at different points in time. As time progresses, the data in the database may change. When a fact is deleted from the database, it remains logically present, but it is no longer part of the current state. Instead, the deleted fact becomes a historical version of the data. The version of the fact is associated with the transaction time of deletion, effectively marking the end of its validity as part of the current database state.

Pure timeslice/timerange query Given a query time point $q.t$ or query time interval $[q.tstart, q.tend]$, retrieve the records in all versions of T which were valid at $q.t$ or some time during $[q.tstart, q.tend]$, respectively, together with their validity intervals.

Range timeslice/timerange query Given a query time point $q.t$ or query time interval $[q.tstart, q.tend]$, an attribute A of T, and a range $[q.Astart, q.Aend]$, retrieve the records r in all versions of T which (1) were valid at $q.time$ or some time during $[q.start, q.end]$, respectively, and (2) satisfy $q.Astart \leq r.A \leq q.Aend$ together with their validity intervals.

Chapter 3

Related Work

In this chapter, we review related work on the queries that are relevant with our work and developed under the specifications of (1) indexing valid time and (2) indexing transaction time; we also briefly present other recent work on temporal data management.

3.1 Valid-time indexing

Valid-time temporal databases store record versions which are valid during a well-defined time interval [38]. This interval could refer to the past, the future, or may start at some time in the past and finish in the future (for example, an activated credit card which expires at some time in the future). The order by which records in a valid-time database are inserted, deleted, or updated is not necessarily related to the validity time of the records.

Managing valid-time records for the evaluation of time-travel queries can then be considered as a case of indexing intervals (i.e., one-dimensional ranges), which is a well-studied problem with lots of previous work [35, 39, 7, 36, 40].

Given a set of S of data intervals, the segment tree [7] sorts the distinct end-points of all data intervals and creates a binary search tree for them. Each leaf v corresponds to an elementary interval $Int(v)$ defined by two consecutive distinct end-points in S. Each non-leaf node interval $Int(v)$ corresponds to the union of elementary intervals in the subtree rooted at v. Each data interval $s \in S$ is assigned to nodes, such that node v includes s iff s covers $Int(v)$, but s does not cover the interval of v's parent. Hence, each data interval is assigned to $\log n$ nodes and the segment tree requires $O(n \log n)$ space. Given a query point q, the tree traverses the path of nodes whose intervals include q and reports all intervals in them (in $O(\log n + K)$ time).

The interval tree [35] defines a center point c, such that the data intervals strictly before c are approximately as many as those strictly after c. Then, c becomes the root of the tree and all data intervals that include c are stored at the root. The left and right subtrees of c are defined recursively. The intervals assigned to each node are sorted based on their begin and based on their end-points (i.e., two sorted lists are defined). Hence, the space complexity of the tree is $O(n)$. Given a point query q, if $q \leq c$, then the intervals in the begin-list of root c are accessed and reported until an s with $s.begin > c$ is found; the left subtree is then searched recursively. If $q > c$, then the intervals in the end-list of c are accessed in reverse order and reported until an s with $s.end < c$ is found; the left subtree is then searched recursively. Since at most one non-result interval is accessed per node, the time complexity is $O(\log n + K)$. The tree can also be used to answer range queries at the same complexity. For example, Figure 3.1 shows a set of 14 intervals $s_1, \ldots, s_{14}$, which are assigned to 7 interval tree nodes and a query interval $q = [q.st, q, end]$. The domain point c corresponding to the tree's root is *contained in* the query interval, hence all intervals in the root are reported and both the left and right children of the root have to be visited recursively. Since the left child's point c_L is before $q.st$, we access the END list from the end and report results until we find an interval s for which $s.end < q.st$; then we access recursively the right child of c_L. This process is repeated symmetrically for the root's right child c_R.

A disk-based version of the interval tree was proposed in [39]. In [39] the goal is to manage efficiently intervals using a relational representation in a relational database management system (RDBMS). The authors aim to use built-in composite indices provided by the RDBMS for lower space complexity. The proposed data structure is three-fold. The primary structure is not materialized physically. Instead, the other

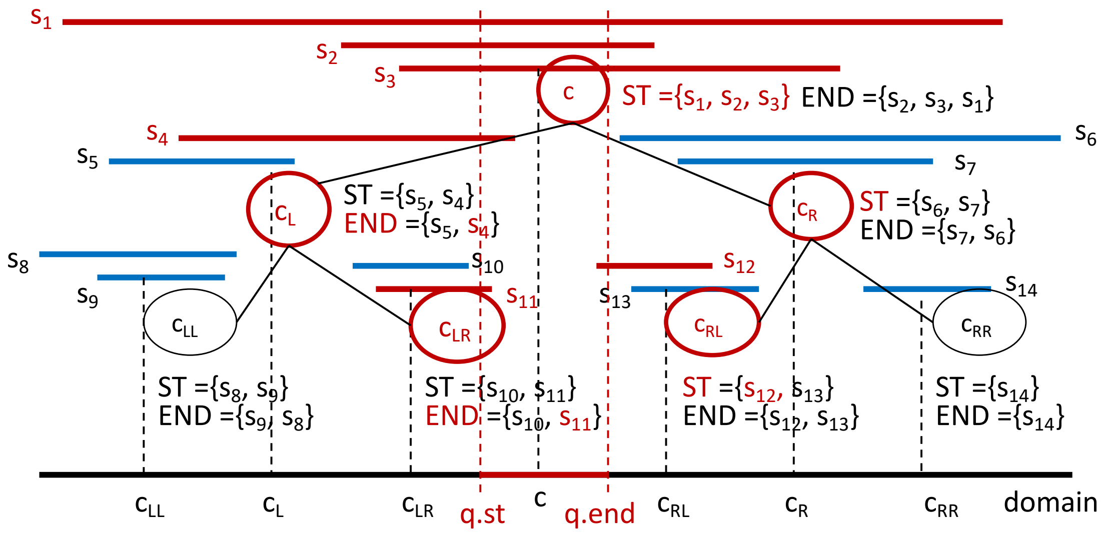


Figure 3.1: Example of an interval tree

two parts of the structure are used to store information for the intervals. The second part of the structure contains a list of lower bounds ($L(W)$) for intervals registered at each node w. This information is represented using the relation $(node, lower)$ in the RDBMS, where $node$ is the identifier and $lower$ is the lower bound of the interval associated with that node. The third part of the structure is linked to the second part and contains the upper bounds in the same way. So, The RDBMS efficiently organizes the two relations (node, lower) and (node, upper) using built-in composite indices (e.g., B+-tree). These indices ensure efficient querying and retrieval of interval data. The resulting relational schema and the composite indices yield a space complexity of $O(n/b)$ for n intervals, where b represents the branching factor of the tree. The technique can be implemented using SQL Data Definition Language (DDL) statements to create the necessary relations and indices in the RDBMS.

A simple and practical data structure for intervals is a *1D-grid*, which divides the domain into p partitions $P_1, P_2, \ldots, P_p$. The partitions are pairwise disjoint in terms of their interval span and collectively cover the entire data domain D. Each interval is assigned to all partitions that it overlaps with. Figure 3.2 shows 5 intervals assigned to $p = 4$ partitions; s_1 goes to P_1 only, while s_5 goes to all four partitions. Given a query q, the results can be obtained by accessing each partition P_i that overlaps with q. For each P_i which is *contained in* q (i.e., $q.st \leq P_i.st \wedge P_i.end \leq q.end$), all intervals in P_i are guaranteed to overlap with q. For each P_i, which overlaps with q, but is not contained in q, we should compare each $s_i \in P_i$ with q to determine whether s_i is a query result. If the interval of a query q overlaps with multiple partitions,

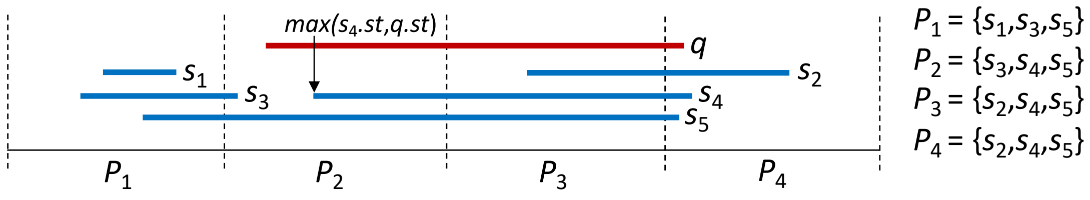


Figure 3.2: Example of a 1D-grid

duplicate results may be produced. An efficient approach for handling duplicates is the *reference value* method [41], which was originally proposed for rectangles but can be directly applied for 1D intervals. For each interval s found to overlap with q in a partition P_i, we compute $v = \max\{s.st, q.st\}$ as the *reference value* and report s only if $v \in [P_i.st, P_i.end]$. Since v is unique, s is reported only in one partition. In Figure 3.2, interval s_4 is reported only in P_2 which contains value $\max\{s_4.st, q.st\}$.

The 1D-grid has two drawbacks. First, the duplicate results should be computed and checked before being eliminated by the reference value. Second, if the collection contains many long intervals, the index may grow large in size due to excessive replication which increases the number of duplicate results to be eliminated. In contrast, 1D-grid supports fast updates as the partitions are stored independently with no need to organize the intervals in them.

Duplicate results can be avoided using the reference point technique [41] or after dividing the data in each partition to classes based on whether they begin inside or before the partition [42]. Data structures for multi-dimensional boxes, such as the R-tree [43, 44], can also be used for 1D intervals. For example, a simple and dynamic data structure for intervals is the 1D-grid, which divides the space into a number of partitions, either uniformly or adaptively to the interval distribution. Each interval is then assigned to all partitions that overlap with it. A point (or range) query q is evaluated by accessing the partition(s) intersecting q and reporting the intervals there after conducting comparisons as necessary.

A data structure which considers both the values and the durations of the intervals is the period index [36]. The *period index* is a self-adaptive structure based on domain partitioning, specialized for `G-OVERLAPS` and *duration* queries. The time domain is split into coarse partitions as in a 1D-grid and then each partition is divided hierarchically, in order to organize the intervals assigned to the partition based on their positions and durations. Figure 3.3 shows a set of intervals and how they are partitioned in a period index. There are two primary partitions P_1 and P_2 and each of them is divided

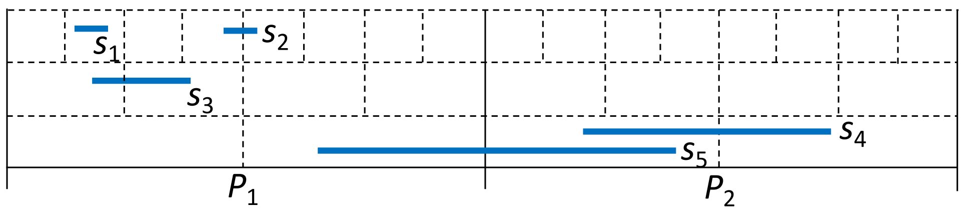


Figure 3.3: Example of a period index

hierarchically to three levels. Each level corresponds to a duration length and each interval is assigned to the level corresponding to its duration. The top level stores intervals shorter than the length of a division there, the second level stores longer intervals but shorter than a division there, and so on. Hence, each interval is assigned to at most two divisions, except for intervals which are assigned to the bottom-most level, which can go to an arbitrary number of divisions. During query evaluation, only the divisions that overlap with the query interval are accessed; if the query carries a duration predicate, the divisions that are shorter than the query duration are skipped. For `G-OVERLAPS` queries, the period index performs in par with the interval tree and the 1D-grid [36], so we also compare against this index in Section 4.3.

An alternative indexing approach is to map intervals to 2D points and then index them by an off-the-shelf spatial data structure [45, 7]. Specifically, each data interval $s = [s.st, s.end)$ is mapped to point $(s.st, s.end)$ in the $D \times D$ space, where D is the domain of the interval endpoints. Figure 3.4 shows a number of intervals as points in this 2D space. Since $s.st < s.end$ for each interval s, the points are all above the diagonal connecting points $(0, 0)$ and (D, D). Each point or range query becomes a rectangular range query in the 2D space, having x- and y-projections $[0, q.end]$ and $[q.start, D]$, respectively, as shown by the shaded rectangle in Figure 3.4. This approach has been used in previous work on managing text document versions [1] and temporal data [2].

Besides, recent research on indexing intervals does not address basic queries such as stabbing or range queries, but more demanding operations such as temporal aggregation [30, 32, 13] and interval joins [25, 26, 27, 28, 29, 46, 47].

Piatov et al. [34] present a collection of plane-sweeping algorithms to extend the timeline index with other forms of temporal aggregation, such as aggregation over fixed intervals, sliding window aggregates, and MIN/MAX aggregates. They apply their methods on the Endpoint Index. The idea behind the Endpoint index is that intervals can be mapped onto one-dimensional endpoints or events.

The lazy endpoint-based interval join algorithm by Piatov et al. [26] is based on the Endpoint Index as well. An in-memory hash map manages the active tuples, which is optimized for sequential reads of the entire map. However, in order to reduce the number of scans on the active tuple map, they pre-allocate the space for the active tuples. When an active tuple is removed from the list, the last inserted active tuple takes its place. This requires the unrealistic assumption that all tuples must be equal-sized.

Furthermore, Bouros and Mamoulis [27] proposed a forward-scan based plane sweep algorithm for temporal joins. They group consecutively sweeped intervals such that join results can be produced in batches in order to avoid redundant comparisons. They also extend the grouping with a bucket index to further reduce the number of comparisons. This work was extended for interval band joins and semi-joins evaluation [48], [49]. More optimization techniques were proposed in [28].

Deficiencies of interval indices. Most interval indexing methods share a weakness: the domain of the interval endpoints should be known apriori. If the data domain grows (i.e., as in a temporal database), the partitions have to potentially be updated to cover the new part of the domain and it might be necessary to change the assignments of data intervals to partitions to maintain the good properties of the index. On the other hand, the 2D point transformation approach [1] does not have this problem as a 2D spatial index such as the R-tree can adapt to a growing domain. Still, the query regions are relatively large and touch a large part of the 2D space, most of which is sparsely populated. More importantly, all methods discussed in this section are not appropriate for indexing *live* data versions in temporal databases, whose end is unknown (i.e., equal to the ever-changing t_{now}). Finally, data structures for intervals are not designed for indexing another attribute at the same time; i.e., they are not appropriate for the *range* time-travel queries discussed in the Chapter 1.

3.2 Transaction-time indexing

Transaction-time databases [50] manage the evolution history of a database. In Chapter 1, we gave an example of such a database containing a table T with employees records. Indexing transaction-time DBs is more challenging than valid-time DBs,

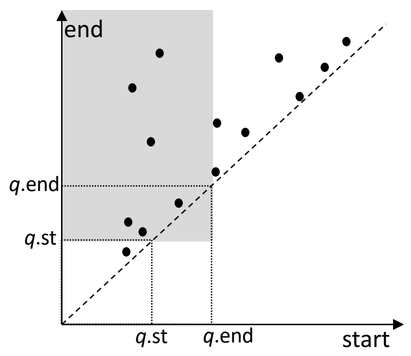
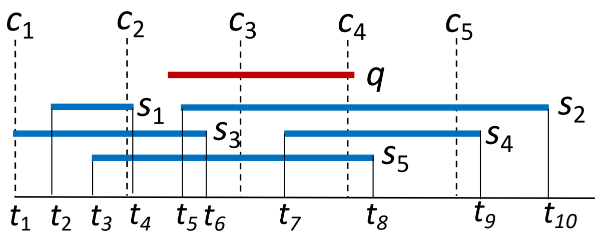


Figure 3.4: 2D mapping [1, 2]

since there are *live* records which are valid now, but we do not know their end-time. These records comprise the current database state and may be changed or deleted in the future, but we are not aware of the exact time for this. On the other hand, *dead* records belong to past states for which we do know their end-time. Records (2, Black, 30K) and (2, Black, 35K) in Figure 1.1 are examples of dead and live records, having validity $[t_0, t_2)$ and $[t_2, t_{now})$, respectively. In fact, these two records correspond to versions of the same record (employee Black). Versions of the same record cannot temporally overlap.

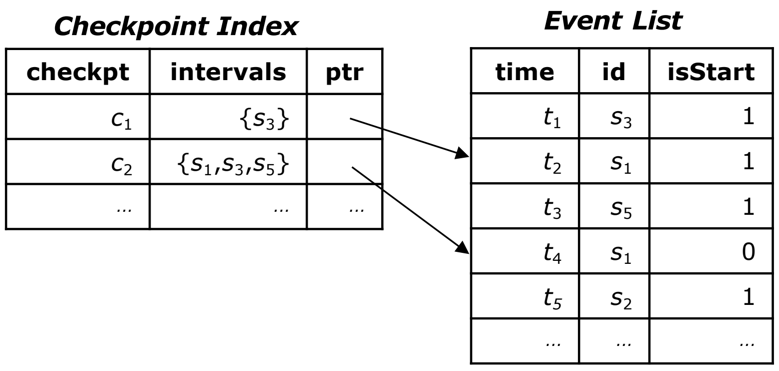
Previous work on temporally indexing an evolving DB table extend current-state indices to support search on all table versions. These indices do not only support pure time travel queries, but also *range* time travel queries based on a search-key attribute A (i.e., from all records r which were valid at some timestamp or period in the past retrieve those for which $v_1 \leq A \leq v_2$). To support such queries, they index simultaneously the temporal versions of the records and their values on the search key attribute A. These methods focus on minimizing disk I/O during search; their main-memory versions are relatively slow in search and updates compared to the interval indices reviewed in Sec. 3.1.

A more recent index for transaction-time DBs implemented in SAP HANA is the *Timeline* index [13], which builds upon the Time index [51] and supports very fast updates. The *timeline index* [13] is a general-purpose access method for temporal (versioned) data, implemented in SAP-HANA. It keeps the endpoints of all intervals in an *event list*, which is a table of $\langle time, id, isStart \rangle$ triples, where $time$ is the value of the start or end point of the interval, id is the identifier of the interval, and $isStart$ 1 or 0, depending on whether $time$ corresponds to the start or end of the interval, respectively. The event list is sorted primarily by $time$ and secondarily by $isStart$ (descending). In addition, at certain timestamps, called *checkpoints*, the entire set of



c_1 c_2 c_3 c_4 c_5
q
s_1 s_2 s_3 s_4 s_5
t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}

(a) set of intervals



Checkpoint Index

checkpt	intervals	ptr
c_1	$\{s_3\}$	
c_2	$\{s_1,s_3,s_5\}$	
...	...	...

Event List

time	id	isStart
t_1	s_3	1
t_2	s_1	1
t_3	s_5	1
t_4	s_1	0
t_5	s_2	1
...	...	...

(b) timeline index

Figure 3.5: Example of a timeline index

active object-ids is materialized, that is the intervals that contain the checkpoint. For each checkpoint, there is a link to the first triple in the event list for which $isStart$=0 and $time$ is greater than or equal to the checkpoint, Figure 3.5(a) shows a set of five intervals $s_1, \ldots, s_5$ and Figure 3.5(b) exemplifies a timeline index for them.

To evaluate a *time-travel* query, we first need to find the largest checkpoint which is smaller than or equal to $q.st$ (e.g., c_2 in Figure 3.5) and initialize R as the active interval set at the checkpoint (e.g., $R = \{s_1, s_3, s_5\}$). Then, we scan the event list from the position pointed by the checkpoint, until the first triple for which $time \geq q.st$, and update R by inserting to it intervals corresponding to an $isStart = 1$ event and deleting the ones corresponding to a $isStart = 0$ triple (e.g., R becomes $\{s_3, s_5\}$). When we reach $q.st$, all intervals in R are guaranteed query results and they are reported. We continue scanning the event list until the first triple after $q.end$ and we add to the result the ids of all intervals corresponding to triples with $isStart = 1$ (e.g., s_2 and s_4).

The timeline index accesses more data and performs more comparisons than necessary, during query evaluation. The index also requires a lot of extra space to store the active sets of the checkpoints. Finally, ad-hoc updates are expensive because the event list should be kept sorted.

Timeline index is a part of an important direction of work in temporal DBs, which are general methods to model and organize temporal data. Salzberg et al.'s survey [24] sheds light on the typical access patterns, oriented towards different query types, which play a crucial role in handling temporal data efficiently. These patterns aid in understanding how various index structures support these fundamental operations.

The time-split B-tree (TSB-tree) [52, 53] optimizes the storage of data on erasable media like magnetic disks while migrating older data to another disk, which could be magnetic or optical. The TSB-tree partitions data in nodes based on both transaction time and attribute, while also separating current records from historical ones.

When a data page becomes full and contains fewer than a certain threshold of distinct alive entries, the TSB-tree performs a split based on transaction time only. This allows for more flexibility in choosing the split time, which can be the "time of last update" rather than just the current time. This automatic migration of older versions of records to a separate historical database ensures efficient management of historical data without manual intervention.

This approach differs from vacuuming in systems like POSTGRES [54], which requires a separate background process to find and remove dead records. Additionally, the TSB-tree's time splitting contrasts with methods that reserve optical pages for immovable pages and maintain two addresses (magnetic and optical) for searching contents.

During time splitting in the TSB-tree, the current page retains its existing contents, while historical records are sequentially written to the optical disk. The new optical disk address and the time of the split are recorded in the parent node of the TSB-tree. The process only affects the node being split, the newly allocated node, and the parent node (with occasional further splitting of a full parent). Regardless of whether the new node is stored on an optical disk or not, a split is necessary since the node is full and is receiving new data.

In [55], the authors describe how to convert an ephemeral B+-tree (where only the new values are retained) into one where none of the old values of the B+- tree are lost due to incoming updates. This is achieved by maintaining different versions of the B+-tree such that when a version of the B+-tree is updated, it gives rise to a new version of the B+-tree. Such a structure is referred to as fully persistent, because all the versions can be accessed or updated.

Full-persistence is achieved by capturing the changes made to a node in the B+-tree at the node itself without throwing away any of the old values. As updates come in, information at a node grows, and hence the nodes are referred to as fat nodes. A fat node in a fully-persistent B+-tree consists of a list with ordered pairs of a version identifier and a pointer to an index block. Whenever a fat node is to be updated, a new entry is made in its version block and a new index block pointed to by this entry holds the updated information. Note that in order to be space efficient, a new version of the B+-tree holds only the incremental changes. The versions are in partial order which is maintained in a DAG called a version graph. A version graph is maintained in conjunction with the versioned B+-tree data structure where an edge from node i to node j exists if version j is obtained by updating version i.

The MVBT (Multi-Version B-Tree) [37] introduces efficient support for updates. Efficiently supporting updates involves node consolidation, a process that helps manage space utilization effectively.

One significant feature of the MVBT is its use of node-copying, a concept proposed by Driscoll et al. [56]. Node-copying allows for more efficient handling of data updates. The MVBT also prevents thrashing, which is an important consideration to maintain the tree's stability and performance. Similar to the persistent B-tree, the MVBT utilizes a root structure. When the root performs a time-split, the sibling node becomes the new root, and a new entry is added to the root, pointing to this new root. If the root does a time-and-attribute split, the resulting tree gains an additional level. Furthermore, if a child of the root becomes sparse and merges with its only sibling, the newly merged node becomes the root of a new tree. When a data node becomes full, a copy is made of all the "alive" records at the time the version makes the update that causes the overflow. If the number of distinct records in the copy exceeds a certain threshold, the copy is split into two nodes based on the attributes.

The MVBT takes a different approach from the persistent B-tree and does not create fat nodes. Instead, during a split, they post information about the split to the parent of the overflowing data node, which leads to the creation of new index entries in the parent node to describe the split. If the split produces only one new data node, the attribute used as the lower limit for the overflowing child is copied to a new index entry. The old child pointer is then updated with the time of the copy as its end time, while the new child pointer gets the split time as its start time. If the split results in two new children, they both have the same start time, but one contains the attribute of the overflowing child, and the other holds the attribute used for the attribute split.

As highlighted, most of the existing index structures were originally developed in the mid-to-late '90s with a primary focus on optimizing hard-disk efficiency. Back then, minimizing the number of I/O operations for updates and queries was paramount due to the limitations of hard-disk access speeds. Consequently, tree indices over intervals or versions were commonly employed, complemented by diverse clustering strategies for time and attribute values, as well as partial replication to enhance overall efficiency.

Despite some proposals, like the MVBT [37], showcasing asymptotically optimal I/O behavior for specific temporal queries based on their well-defined objectives, it is important to recognize that the landscape of data storage and processing has evolved significantly since the '90s. Today, the predominant shift towards main-memory settings has introduced new tradeoffs between access time, transfer speeds, and CPU cost. As a result, the performance of these traditional index structures might not be fully optimized in modern, memory-driven architectures.

Considering the previous work, further research is required to adapt and refine existing index structures to suit the challenges and opportunities of main-memory environments. Exploring innovative ways to leverage parallelization and distribution techniques could lead to enhanced efficiency and improved performance. By enriching the study of temporal data organization with new methodologies and insights, more effective and scalable solutions can be created that align with the current data management landscape.

3.3 Other related work

Recent work in temporal databases studies the efficient evaluation of other queries, besides time-travel selections. *Temporal aggregation* [30, 31, 32, 33, 34] computes aggregates of valid record versions (e.g., total project funding) during a query time period (e.g., from 3-23-2021 to 5-15-2023); the output is one value for each time interval in the query period where the aggregate does not change. *Temporal top-k* queries [1, 16] are a special case of temporal aggregation. A *temporal join* [27, 15, 57, 21] finds pairs of record versions (in two different tables) whose validities temporally overlap and they agree on the join key attribute.

In the Overlap Interval Partitioning (OIP) approach [25], the authors divide the time domain into equal-sized granules, create partitions with increasing length that span the entire time domain, and insert each tuple into the shortest partition into which the tuple fits. The join is computed by identifying for each outer partition the overlapping inner partitions which is very efficient. Although, a nested-loop is used for joining the partitions with overlapping tuples.

Another partitioning based approach is the Disjoint Interval Partitioning (DIP) [29]. The main idea behind DIP is to divide each of the two input relations into partitions, such that each partition contains only disjoint intervals. In that way, every partition of one input can be joined with all of the other without backtracking since intervals in the same partition do not overlap. These approaches focus on reducing the search space within a temporal join. Historical *what-if* queries compute the effect that a change in a historical record value would have to the evolution of the database [20].

Other recent related work includes the definition of new temporal semantics [14], system optimizations in the implementation of temporal and multi-version databases [2, 17, 19], temporal database benchmarking [18], and novel temporal integrity constraints [22]. Furthermore, Gutierrez et al. [58] presented a framework to incorporate temporal reasoning into RDF. Bereta et al. [59] implemented valid time component of stRDF and stSPARQL in Strabon. Telos [60], a knowledge representation language, explicitly represents time, which can be crucial for modeling temporal aspects of information systems, such as scheduling and event sequencing.

Major DBMS providers include temporal capabilities into their database management systems. IBM DB2 [61] supports both valid time (referred to as business time) and transaction time (referred to as system time) tables. Queries can target both aspects of the time. Teradata [62] and Oracle DBMS [63] have integrated temporal features with special data types (e.g. *PERIOD*) and functions. Additionally, they have added functionality for managing both valid time and transaction time. Originally, PostgreSQL [64] introduced an external module that brought in a *PERIOD* data type for intervals, along with Boolean predicates and functions. Later, the functionality was integrated into the core of PostgreSQL [65], focusing on range types, designed to represent generic intervals, including associated predicates, functions, and indices. For example, GiST (Generalized Search Tree) is an extensible index which supports querying range types.

Chapter 4

Indexing Intervals

In this chapter, we propose a novel and general-purpose Hierarchical index for INTervals (HINT), which is not suitable just for valid-time data in temporal databases, but for any applications that manage large collections of intervals.

Outline The rest of the chapter is organized as follows. In Section 4.1, we present HINT and its generalized HINT^m version and analyze their complexity. Optimizations that boost the performance of HINT^m are presented in Section 4.2. We evaluate the performance of HINT^m experimentally in Section 4.3 on real and synthetic data and compare it to the state-of-the-art. Finally, Section 4.4 concludes the chapter.

4.1 HINT

In this section, we propose the *Hierarchical index for INTervals* or HINT, which 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

Table 4.1: Table of notation

notation	**description**
$s.id, s.st, s.end$	identifier, start, end point of interval s
$q = [q.st, q.end]$	query interval
$prefix(k, x)$	k-bit prefix of integer x
$P_{\ell,i}$	i-th partition at level ℓ of HINT/HINT^m
$P_{\ell,f}$ ($P_{\ell,l}$)	first (last) partition at level ℓ overlapping with q
$P^O_{\ell,i}$ ($P^R_{\ell,i}$)	sub-partition of $P_{\ell,i}$ with originals (replicas)
$P^{O_{in}}_{\ell,i}$ ($P^{O_{aft}}_{\ell,i}$)	intervals in $P^O_{\ell,i}i$ ending inside (after) $P_{\ell,i}$
$P^{R_{in}}_{\ell,i}$ ($P^{R_{aft}}_{\ell,i}$)	intervals in $P^R_{\ell,i}i$ ending inside (after) $P_{\ell,i}$

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 query results and minimizes the number of accessed intervals. In Section 4.1.1, we present a version of HINT, which avoids comparisons overall during query evaluation, but it is not always applicable and may have high space requirements. Section 4.1.2 presents HINT^m, the general version of our index, used for intervals in arbitrary domains. Last, Section 4.1.3 describes our analytical model for setting the m parameter and Section 4.1.4 discusses updates. Table 4.1 summarizes the notation used in the paper.

4.1.1 A comparison-free version of HINT

We first describe a version of HINT, which is appropriate in the case of a *discrete* and *not very large* domain D. Specifically, assume that the domain D wherefrom the endpoints of intervals in $\mathcal{S}$ take value is $[0, 2^m{-}1]$. We can define 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 $P_{\ell,0}, \ldots, P_{\ell,2^\ell-1}$. Figure 4.1 illustrates the 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 4.1, interval $[5, 9]$ is assigned to one partition at level $\ell = 4$ and two partitions at level $\ell = 3$. The assignment procedure is described by Algorithm 4.1. 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), we assign the interval to partition $P_{\ell,a}$ (resp. $P_{\ell,b}$) and increase a (resp. decrease b) by one. We then update a and b by cutting-off

Algorithm 4.1 Assignment of an interval to partitions

```
Input    : HINT index $\mathcal{H}$, interval $s$
Output   : updated $\mathcal{H}$ after indexing $s$
$a \leftarrow s.st$; $b \leftarrow s.end$;                    ▷ set masks to $s$ endpoints
$\ell \leftarrow m$;                                          ▷ start at the bottom-most level
while $\ell \geq 0$ and $a \leq b$ do
    if last bit of $a$ is 1 then
        add $s$ to $\mathcal{H}.P_{\ell,a}$;                  ▷ update partition
        $a \leftarrow a + 1$
    end
    if last bit of $b$ is 0 then
        add $s$ to $\mathcal{H}.P_{\ell,b}$;                  ▷ update partition
        $b \leftarrow b - 1$
    end
    $a \leftarrow a \div 2$; $b \leftarrow b \div 2$;         ▷ cut-off last bit
    $\ell \leftarrow \ell - 1$;                               ▷ repeat for previous level
end
```

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.

Query evaluation

A selection 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 $P_{\ell,prefix(\ell,q.st)}$ to $P_{\ell,prefix(\ell,q.end)}$, where $prefix(k, x)$ denotes the k-bit prefix of integer x. We call these partitions *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.

We propose 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, we divide the intervals in each partition $P_{\ell,i}$ into two groups: *originals* $P^O_{\ell,i}$ and *replicas* $P^R_{\ell,i}$. Group $P^O_{\ell,i}$ contains all intervals $s \in P_{\ell,i}$ that *begin* at $P_{\ell,i}$ i.e.,

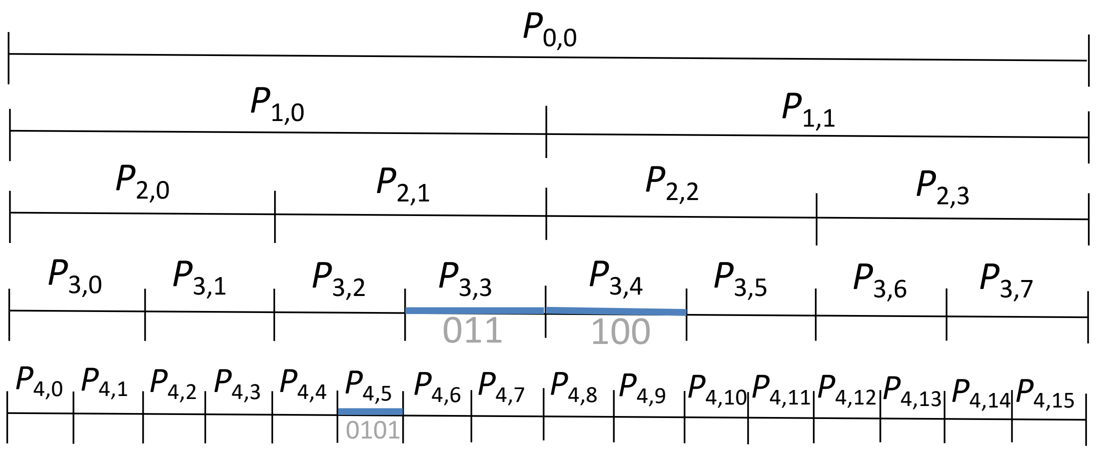

$P_{0,0}$
$P_{1,0}$
$P_{1,1}$
$P_{2,0}$
$P_{2,1}$
$P_{2,2}$
$P_{2,3}$
$P_{3,0}$ $P_{3,1}$ $P_{3,2}$ $P_{3,3}$ $P_{3,4}$ $P_{3,5}$ $P_{3,6}$ $P_{3,7}$
011
100
$P_{4,0}$ $P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{4,4}$ $P_{4,5}$ $P_{4,6}$ $P_{4,7}$ $P_{4,8}$ $P_{4,9}$ $P_{4,10}$ $P_{4,11}$ $P_{4,12}$ $P_{4,13}$ $P_{4,14}$ $P_{4,15}$
0101

Figure 4.1: Hierarchical partitioning and assignment of $[5, 9]$

$prefix(\ell, s.st) = i$. Group $P^R_{\ell,i}$ contains all intervals $s \in P_{\ell,i}$ that begin before $P_{\ell,i}$, i.e., $prefix(\ell, s.st) \neq i$.[1] Each interval is added as original in only one partition of HINT. For example, interval $[5, 9]$ in Figure 4.1 is added to $P^O_{4,5}$, $P^R_{3,3}$, and $P^R_{3,4}$.

Given a query q, at each level ℓ of the index, we report all intervals in the first relevant partition $P_{\ell,f}$ (i.e., $P^O_{\ell,f} \cup P^R_{\ell,f}$). Then, for every other relevant partition $P_{\ell,i}$, $i > f$, we report all intervals in $P^O_{\ell,i}$ and ignore $P^R_{\ell,i}$. 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 $P^R_{\ell,f}$ 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 $P^R_{\ell,i}$ in subsequent relevant partitions ($i > f$) contain intervals, which are either originals in a previous partition $P_{\ell,j}$, $j < i$ or replicas in $P^R_{\ell,f}$, so, they can safely be skipped. Algorithm 4.2 describes the search algorithm using HINT.

For example, consider the hierarchical partitioning of Figure 4.2 and a query interval $q = [5, 9]$. 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 $P_{\ell,f}$ (bold/blue) are reported while in the subsequent partitions (dashed/red), *only* the *original* intervals are.

Discussion. The version of HINT described above finds all query results, without

[1]Whether an interval $s \in P_{\ell,i}$ is assigned to $P^O_{\ell,i}$ or $P^R_{\ell,i}$ is determined at insertion time (Algorithm 4.1). At the first time Line 5 is executed, s is added as an original and in all other cases as a replica. If Line 5 is never executed, then s is added as original the only time that Line 8 is executed.

Algorithm 4.2 Searching HINT

```
Input   : HINT index $\mathcal{H}$, query interval $q$
Output  : set $\mathcal{R}$ of all intervals that overlap with $q$
$\mathcal{R} \leftarrow \emptyset$
foreach level $\ell$ in $\mathcal{H}$ do
    $p \leftarrow prefix(\ell, q.st)$
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,p} \cup \mathcal{H}.P^R_{\ell,p}\}$
    while $p < prefix(\ell, q.end)$ do
        set $p \leftarrow p + 1$
        $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,p}\}$
    end
end
return $\mathcal{R}$
```

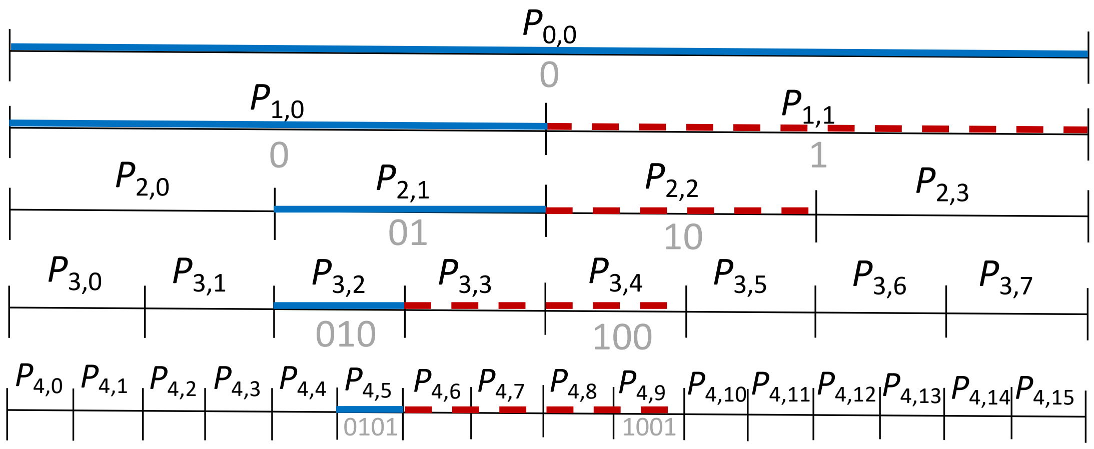


Figure 4.2: Accessed partitions for query $[5, 9]$

conducting any comparisons. This means that in each partition $P_{\ell,i}$, we only have to keep the ids of the intervals that are assigned to $P_{\ell,i}$ 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, we should first normalize all interval endpoints by subtracting the minimum endpoint, in order to convert them to values in a $[0, 2^m - 1]$ domain (the same transformation should be applied on the queries). If the required m is very large, we can index the intervals based on their m-bit prefixes and support approximate search on discretized data. Approximate search can also be applied on intervals in a real-valued domain, after rescaling and discretization in a similar way.

4.1.2 HINT^m: indexing arbitrary intervals

We now present a generalized version of HINT, denoted by HINT^m, which can be used for intervals in arbitrary domains. HINT^m 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 $f(\mathbb{R} \rightarrow D)$ is $f(x) = \lfloor \frac{x - min(x)}{max(x) - min(x)} \cdot (2^m - 1) \rfloor$, 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 HINT^m, using Algorithm 4.1.

For the ease of presentation, we will 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 4.1. So, in contrast to HINT, the set of partitions whereto an interval s is assigned in HINT^m 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 $P_{\ell,i}$ to $P^O_{\ell,i}$ and $P^R_{\ell,i}$, to avoid duplicate results.

Query evaluation using HINT^m

For a query q, simply reporting all intervals in the relevant partitions at each level (as in Algorithm 4.2) 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, we can consider each level of HINT^m as a 1D-grid (see Chapter 3) and go through the partitions at each level ℓ that overlap with q. For the first partition $P_{\ell,f}$, we verify whether s overlaps with q for each interval $s \in P^O_{\ell,f}$ and each $s \in P^R_{\ell,f}$. For the last partition $P_{\ell,l}$, we verify whether s overlaps with q for each interval $s \in P^O_{\ell,l}$. For each partition $P_{\ell,i}$ between $P_{\ell,f}$ and $P_{\ell,l}$, we report all $s \in P^O_{\ell,i}$ without any comparisons. As an example, consider the HINT^m index and the query interval q shown in Figure 4.3. The identifiers of the relevant partitions to q are shown in the figure (and also some indicative intervals that are assigned to these partitions). At level $m = 4$, we have to perform comparisons 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

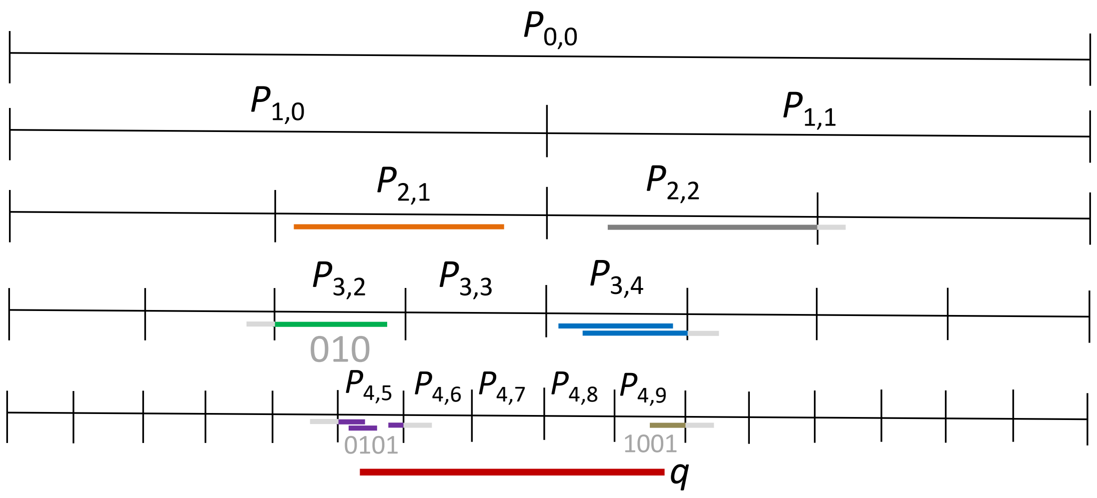


Figure 4.3: Avoiding redundant comparisons in HINT^m

them as results, while in partition $P_{4,9}$ we compare the start points of all originals with q, before we can confirm whether they are results or not. We can simplify the overlap tests at the first and the last partition of each level ℓ based on the following:

Lemma 4.1. *At every level ℓ, each $s \in P^R_{\ell,f}$ is a query result iff $q.st \le s.end$. If $l > f$, each $s \in P^O_{\ell,l}$ is a query result iff $s.st \le q.end$.*

Proof. For the first relevant partition $P_{\ell,f}$ at each level ℓ, for each replica $s \in P^R_{\ell,f}$, $s.st < q.st$, so $q.st \le s.end$ suffices as an overlap test. For the last partition $P_{\ell,l}$, if $l > f$, for each original $s \in P^O_{\ell,f}$, $q.st < s.st$, so $s.st \le q.end$ suffices as an overlap test. □

Avoiding redundant comparisons in query evaluation

One of our most important findings in this study and a powerful feature of HINT^m 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, we do not have to perform comparisons 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 point 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 $s.st \le q.end$ tests as intervals in these partitions are guaranteed to start before $P_{4,9}$. The lemma below formalizes these observations:

Lemma 4.2. *If the first (resp. last) relevant partition for a query q at level ℓ ($\ell < m$) starts (resp. ends) at the same value as the first (resp. last) relevant partition at level $\ell + 1$, then for every first (resp. last) relevant partition $P_{v,f}$ (resp. $P_{v,l}$) at levels $v \leq \ell$, each interval $s \in P_{v,f}$ (resp. $s \in P_{v,l}$) satisfies $s.end \geq q.st$ (resp. $s.st \leq q.end$).*

Proof. Let $P.st$ (resp. $P.end$) denote the first (resp. last) domain value of partition P. Consider the first relevant partition $P_{\ell,f}$ at level ℓ and assume that $P_{\ell,f}.st = P_{\ell+1,f}.st$. Then, for every interval $s \in P_{\ell,f}$, $s.end \geq P_{\ell+1,f}.end$, otherwise s would have been allocated to $P_{\ell+1,f}$ instead of $P_{\ell,f}$. Further, $P_{\ell+1,f}.end \geq q.st$, since $P_{\ell+1,f}$ is the first partition at level $\ell+1$ which overlaps with q. Hence, $s.end \geq q.st$. Moreover, for every interval $s \in P_{v,f}$ with $v < \ell$, $s.end \geq P_{\ell+1,f}.end$ holds, as interval $P_{v,f}$ covers interval $P_{\ell,f}$; so, we also have $s.end \geq q.st$. Symmetrically, we prove that if $P_{\ell,l}.end = P_{\ell+1,l}.end$, then for each $s \in P_{v,l}, v \leq \ell$, $s.st \leq q.end$. □

We next focus on how to rapidly check the condition of Lemma 4.2. Essentially, if the last bit of the offset f (resp. l) of the first (resp. last) partition $P_{\ell,f}$ (resp. $P_{\ell,l}$) relevant to the query at level ℓ is 0 (resp. 1), then the first (resp. last) partition at level $\ell - 1$ above satisfies the condition. For example, in Figure 4.3, consider the last relevant partition $P_{4,9}$ at level 4. The last bit of $l = 9$ is 1; so, the last partition $P_{3,4}$ at level 3 satisfies the condition and we do not have to perform comparisons in the last partitions at level 3 and above.

Algorithm 4.3 is a pseudocode for the search algorithm on HINT^m. 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, according to the discussion in the previous paragraph. At each level ℓ, we find the offsets of the relevant partitions to the query, based on the ℓ-prefixes of $q.st$ and $q.end$. For the first position $f = prefix(q, st)$, the partitions holding originals and replicas $P^O_{\ell,f}$ and $P^R_{\ell,f}$ 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 we perform all comparisons in $P^O_{\ell,f}$ and apply the first observation in Lemma 4.1 to $P^R_{\ell,f}$. Else, if only $complast$ is set, we can safely skip the $q.st \leq s.end$ comparisons; if only $compfist$ is set, regardless whether $f = l$, we just perform $q.st \leq s.end$ comparisons to both originals and replicas to the first partition. Finally, if neither $compfirst$ nor $complast$ are set, we just report all intervals in the first partition as results. If we are at the last partition $P_{\ell,l}$ and $l > f$ (Line 17) then we just examine $P^O_{\ell,l}$ and apply just the $s.st \leq q.end$ test for each interval there, according to Lemma 4.1. Finally, for all partitions in-between the first and the last one, we simply report all original intervals there.

Complexity Analysis

Let n be the number of intervals in $\mathcal{S}$. Assume that the domain is $[0, 2^{m'} - 1]$, where $m' > m$. To analyze the space complexity of HINT^m, we first prove the following:

Lemma 4.3. *The total number of intervals assigned at the lowest level m of HINT^m is expected to be n.*

Proof. Each interval $s \in \mathcal{S}$ will go to zero, one, or two partitions at level m, based on the bits of $s.st$ and $s.end$ at position m (see Algorithm 4.1); on average, s will go to one partition. $\square$

Using Algorithm 4.1, when an interval is assigned to a partition at a level ℓ, the interval is *truncated* (i.e., shortened) by $2^{m'-\ell}$. Based on this, we analyze the space complexity of HINT^m as follows.

Theorem 4.1. *Let λ be the average length of intervals in input collection S. The space complexity of HINT^m is $O(n \cdot \log_2(2^{\log_2 \lambda - m' + m} + 1))$.*

Proof. Based on Lemma 4.3, each $s \in S$ will be assigned on average to one partition at level m and will be truncated by $2^{m'-m}$. Following Algorithm 4.1, at the next level $m - 1$, s is also be expected to be assigned to one partition (see Lemma 4.3) and truncated by $2^{m'-m+1}$, and so on, until the entire interval is truncated (condition $a \leq b$ is violated at Line 3 of Algorithm 4.1). Hence, we are looking for the number of levels whereto each s will be assigned, or for the smallest k for which $2^{m'-m} + 2^{m'-m+1} + \cdots + 2^{m'-m+k-1} \geq \lambda$. Solving the inequality gives $k \geq \log_2(2^{\log_2 \lambda - m' + m} + 1)$ and the space complexity of HINT^m is $O(n \cdot k)$ $\square$

For the computational cost of queries in terms of conducted comparisons, in the worst case, $O(n)$ intervals are assigned to the first relevant partition $P_{m,f}$ at level m and $O(n)$ comparisons are required. To estimate the *expected cost* of query evaluation in terms of conducted comparisons, we assume a uniform distribution of intervals to partitions and random query intervals.

Lemma 4.4. *The expected number of HINT^m partitions for which we have to conduct comparisons is four.*

Proof. At the last level of the index m, we definitely have to do comparisons in the first and the last partition (which are different in the worst case). At level $m-1$, for each of the first and last partitions, we have a 50% chance to avoid comparisons, due to Lemma 4.2. Hence, the expected number of partitions for which we have to perform comparisons at level $m-1$ is 1. Similarly, at level $m-2$ each of the yet active first/last partitions has a 50% chance to avoid comparisons. Overall, for the worst-case conditions, where m is large and q is long, the expected number of partitions, for which we need to perform comparisons is $2+1+0.5+0.25+\cdots=4$. □

Theorem 4.2. *The expected number of comparisons during query evaluation over HINT^m is $O(n/2^m)$.*

Proof. For each query, we conduct comparisons at least in the first and the last relevant partitions at level m. The expected number of intervals, in each of these two partitions, is $O(n/2^m)$, considering Lemma 4.3 and assuming a uniform distribution of the intervals in the partitions. In addition, due to Lemma 4.4, the number of expected additional partitions that require comparisons is 2 and each of these two partitions is expected to also hold at most $O(n/2^m)$ intervals, by Lemma 4.3 on the levels above m and using the truncated intervals after their assignment to level m (see Algorithm 4.1). Hence, q is expected to be compared with $O(n/2^m)$ intervals in total and the cost of each such comparison is $O(1)$. □

4.1.3 Setting m

As shown in Section 4.1.2, the space requirements and the search performance of HINT^m depend on the value of m. For large values of m, the cost of accessing comparison-free results will dominate the computational cost of comparisons. We conduct an analytical study for estimating m_{opt}: the smallest value of m, which is expected to result in a HINT^m of search performance close to the best possible, while achieving the lowest possible space requirements. Our study uses simple statistics namely, the number of intervals $n=|\mathcal{S}|$, the mean length λ_s of data intervals and the mean length λ_q of query intervals. We assume that the endpoints and the lengths of intervals and queries are uniformly distributed.

The overall cost of query evaluation consists of (1) the cost for determining the relevant partitions per level, denoted by C_p, (2) the cost of conducting comparisons between data intervals and the query, denoted by C_{cmp}, and (3) the cost of accessing query results in the partitions for which we do not have to conduct comparisons, denoted by C_{acc}. Cost C_p is negligible, as the partitions are determined by a small number m of bit-shifting operations. To estimate C_{cmp}, we need to estimate the number of intervals in the partitions whereat we need to conduct comparisons and multiply this by the expected cost β_{cmp} per comparison. To estimate C_{acc}, we need to estimate the number of intervals in the corresponding partitions and multiply this by the expected cost β_{acc} of (sequentially) accessing and reporting one interval. β_{cmp} and β_{acc} are machine-dependent and can easily be estimated by experimentation.

According to Algorithm 4.3, unless λ_q is smaller than the length of a partition at level m, there will be two partitions that require comparisons at level m, one partition at level $m-1$, etc. with the expected number of partitions being at most four (see Lemma 4.4). Hence, we can assume that C_{cmp} is practically dominated by the cost of processing two partitions at the lowest level m. As each partition at level m is expected to have $n/2^m$ intervals (see Lemma 4.3), we have $C_{cmp} = \beta_{cmp} \cdot n/2^m$. Then, the number of accessed intervals for which we expect to apply no comparisons is $|Q| - 2 \cdot n/2^m$, where $|Q|$ is the total number of expected query results. Under this, we have $C_{acc} = \beta_{acc} \cdot (|Q| - 2 \cdot n/2^m)$. We can estimate $|Q|$ using the selectivity analysis for (multidimensional) intervals and queries in [66] as $|Q| = n \cdot \frac{\lambda_s + \lambda_q}{\Lambda}$, where Λ is the length of the entire domain with all intervals in $\mathcal{S}$ (i.e., $\Lambda = \max_{\forall s \in \mathcal{S}} s.end - \min_{\forall s \in \mathcal{S}} s.st$).

With C_{cmp} and C_{acc}, we now discuss how to estimate m_{opt}. First, we gradually increase m from 1 up to its max value m' (determined by Λ), and compute the expected cost $C_{cmp} + C_{acc}$. For $m = m'$, HINT^m corresponds to the comparison-free HINT with the lowest expected cost. Then, we select as m_{opt} the lowest value of m for which $C_{cmp} + C_{acc}$ converges to the cost of the $m = m'$ case.

4.1.4 Updates

We handle insertions to an existing HINT or HINT^m index by calling Algorithm 4.1 for each new interval s. Small adjustments are needed for HINT^m to add s to the originals division at the first partition assignment, i.e., to $P^O_{\ell,a}$ or $P^O_{\ell,b}$, and to the replicas division for every other partition, i.e., to $P^R_{\ell,a}$ or $P^R_{\ell,b}$ Finally, we handle deletions using tombstones, similarly to previous studies [67, 68] and recent indexing approaches [69]. Given an interval s for deletion, we first search the index to locate all partitions that contain s (both as original and as replica) and then, replace the id of s by a special "tombstone" id, which signals the logical deletion.

4.2 Optimizing HINT^m

In this section, we discuss optimization techniques, which greatly improve the performance of HINT^m (and HINT) in practice. First, we show how to reduce the number of partitions in HINT^m where comparisons are performed and how to avoid accessing unnecessary data. Next, we show how to handle very sparse or skewed data at each level of HINT/HINT^m. Another optimization is decoupling the storage of the interval ids with the storage of interval endpoints in each partition. Finally, we revisit updates under the prism of these optimizations.

4.2.1 Subdivisions and space decomposition

Recall that, at each level ℓ of HINT^m, every partition $P_{\ell,i}$ is divided into $P^O_{\ell,i}$ (holding originals) and $P^R_{\ell,i}$ (holding replicas). We propose to further divide each $P^O_{\ell,i}$ into $P^{O_{in}}_{\ell,i}$ and $P^{O_{aft}}_{\ell,i}$, so that $P^{O_{in}}_{\ell,i}$ (resp. $P^{O_{aft}}_{\ell,i}$) holds the intervals from $P^{O_{in}}_{\ell,i}$ that end *inside* (resp. *after*) partition $P_{\ell,i}$. Similarly, each $P^R_{\ell,i}$ is divided into $P^{R_{in}}_{\ell,i}$ and $P^{R_{aft}}_{\ell,i}$.

Queries that overlap with multiple partitions. Consider a query q, which overlaps with a sequence of *more than one partitions* at level ℓ. As already discussed, if we have to conduct comparisons in the first such partition $P_{\ell,f}$, we should do so for all intervals in $P^O_{\ell,f}$ and $P^R_{\ell,f}$. By subdividing $P^O_{\ell,f}$ and $P^R_{\ell,f}$, we get the following lemma:

Lemma 4.5. *If $P_{\ell,f} \neq P_{\ell,l}$ (1) each interval s in $P^{O_{in}}_{\ell,f} \cup P^{R_{in}}_{\ell,f}$ overlaps with q iff $s.end \geq q.st$; and (2) all intervals s in $P^{O_{aft}}_{\ell,f}$ and $P^{R_{aft}}_{\ell,f}$ surely overlap with q.*

Proof. Follows directly from the fact that q starts *inside* $P_{\ell,f}$ but ends *after* $P_{\ell,f}$. □

Hence, we need *just one comparison* for each interval in $P^{O_{in}}_{\ell,f} \cup P^{R_{in}}_{\ell,f}$, whereas we can report all intervals $P^{O_{aft}}_{\ell,f} \cup P^{R_{aft}}_{\ell,f}$ as query results *without any comparisons*. As already discussed, for all partitions $P_{\ell,i}$ between $P_{\ell,f}$ and $P_{\ell,l}$, we just report intervals

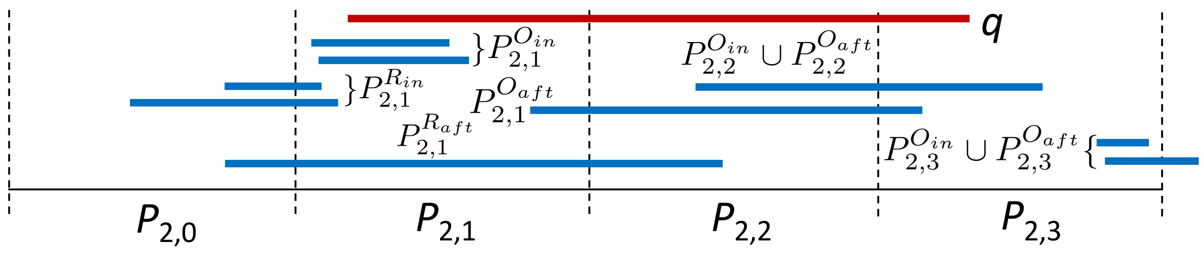


Figure 4.4: Partition subdivisions in HINT^m (level $\ell = 2$)

in $P_{\ell,i}^{O_{in}} \cup P_{\ell,i}^{O_{aft}}$ as results, without any comparisons, whereas for the last partition $P_{\ell,l}$, we perform *one comparison* per interval in $P_{\ell,l}^{O_{in}} \cup P_{\ell,l}^{O_{aft}}$.

Queries that overlap with a single partition. If the query q overlaps only one partition $P_{\ell,f}$ at level ℓ, we can use following lemma to minimize the necessary comparisons:

Lemma 4.6. *If $P_{\ell,f} = P_{\ell,l}$ then*

- *each interval s in $P_{\ell,f}^{O_{in}}$ overlaps with q iff $s.st \leq q.end \wedge q.st \leq s.end$,*
- *each interval s in $P_{\ell,f}^{O_{aft}}$ overlaps with q iff $s.st \leq q.end$,*
- *each interval s in $P_{\ell,f}^{R_{in}}$ overlaps with q iff $s.end \geq q.st$,*
- *all intervals in $P_{\ell,f}^{R_{aft}}$ overlap with q.*

Proof. All intervals $s \in P_{\ell,f}^{O_{aft}}$ end after q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in P_{\ell,f}^{R_{in}}$ start before q, so $s.st \leq q.end$ suffices as an overlap test. All intervals $s \in P_{\ell,f}^{R_{aft}}$ start before and end after q, so they are guaranteed results. □

Overall, the subdivisions help us to minimize the number of intervals in each partition, for which we have to apply comparisons. Figure 4.4 shows the subdivisions which are accessed by query q at level $\ell = 2$ of a HINT^m index. In partition $P_{\ell,f} = P_{2,1}$, all four subdivisions are accessed, but comparisons are needed only for intervals in $P_{2,1}^{O_{in}}$ and $P_{2,1}^{R_{in}}$. In $P_{2,2}$, only the originals (in $P_{2,2}^{O_{in}}$ and $P_{2,2}^{O_{aft}}$) are accessed and reported without any comparisons. Finally, in $P_{\ell,l} = P_{2,3}$, only the originals (in $P_{2,3}^{O_{in}}$ and $P_{2,3}^{O_{aft}}$) are accessed and compared to q.

Sorting the intervals in each subdivision

We can keep the intervals in each subdivision sorted, in order to reduce the number of comparisons for queries that access them. For example, let us examine the last

Table 4.2: Necessary data and beneficial sort orders

subdivision	beneficial sorting	necessary data
$P^{O_{in}}_{\ell,i}$	by $s.st$ or by $s.end$	$s.id, s.st, s.end$
$P^{O_{aft}}_{\ell,i}$	by $s.st$	$s.id, s.st$
$P^{R_{in}}_{\ell,i}$	by $s.end$	$s.id, s.end$
$P^{R_{aft}}_{\ell,i}$	no sorting	$s.id$

partition $P_{\ell,l}$ that overlaps with a query q at a level ℓ. If the intervals s in $P^{O_{in}}_{\ell,l}$ are sorted on their start endpoint (i.e., $s.st$), we can simply access and report the intervals until the first $s \in P^{O_{in}}_{\ell,l}$, such that $s.st > q.end$. Or, we can perform binary search to find the first $s \in P^{O_{in}}_{\ell,l}$, such that $s.st > q.end$ and then scan and report all intervals before s. Table 4.2 (second column) summarizes the sort orders for each of the four subdivisions of a partition that can be beneficial in query evaluation. For a subdivision $P^{O_{in}}_{\ell,i}$, intervals may have to be compared based on their start point (if $P_{\ell,i} = P_{\ell,f}$), or based on their end point (if $P_{\ell,i} = P_{\ell,l}$), or based on both points (if $P_{\ell,i} = P_{\ell,f} = P_{\ell,l}$). Hence, we choose to sort based on either $s.st$ or $s.end$ to accommodate two of these three cases. For a subdivision $P^{O_{aft}}_{\ell,i}$, intervals may have to be compared *only* based on their start point (if $P_{\ell,i} = P_{\ell,l}$). For a subdivision $P^{R_{in}}_{\ell,i}$, intervals may have to be compared *only* based on their end point (if $P_{\ell,i} = P_{\ell,f}$). Last, for a subdivision $P^{R_{aft}}_{\ell,i}$, there is never any need to compare the intervals, so, no order provides any benefit.

Storage optimization

So far, we have assumed that each interval s is stored in the partitions whereto s is assigned as a triplet $\langle s.id, s.st, s.end \rangle$. However, if we split the partitions into subdivisions, we do not need to keep all information of the intervals in them. Specifically, for each subdivision $P^{O_{in}}_{\ell,i}$, we may need to use $s.st$ and/or $s.end$ for each interval $s \in P^{O_{in}}_{\ell,i}$, while for each subdivision $P^{O_{aft}}_{\ell,i}$, we may need to use $s.st$ for each $s \in P^{O_{in}}_{\ell,i}$, but we will never need $s.end$. From the intervals s of each subdivision $P^{R_{in}}_{\ell,i}$, we may need $s.end$, but we will never use $s.st$. Finally, for each subdivision $P^{R_{aft}}_{\ell,i}$, we just have to keep the $s.id$ identifiers of the intervals. Table 4.2 (third column) summarizes the data that we need to keep from each interval in the subdivisions of each partition. Since each interval s is stored as original just once in the entire index, but as replica in possibly multiple partitions, space can be saved by storing only the necessary data, especially if the intervals span multiple partitions. Note that even when we do not apply the subdivisions, but just use $P^{O}_{\ell,i}$ and $P^{R}_{\ell,i}$ (as suggested in Section 4.1.2), we do not need to store the start points $s.st$ of all intervals in $P^R_{\ell,i}$, as they are never used in comparisons.

4.2.2 Handling data skewness and sparsity

Data skewness and sparsity may cause many partitions to be empty, especially at the lowest levels of HINT (i.e., large values of ℓ). Recall that a query accesses a sequence of multiple $P^O_{\ell,i}$ partitions at each level ℓ. Since the intervals are physically distributed in the partitions, this results into the unnecessary accessing of empty partitions and may cause cache misses. We propose a storage organization where all $P^O_{\ell,i}$ divisions at the same level ℓ are merged into a single table T^O_ℓ and an auxiliary index is used to find each non-empty division.[2] The auxiliary index locates the first non-empty partition, which is greater than or equal to the ℓ-prefix of $q.st$ (i.e., via binary search or a binary search tree). From thereon, the nonempty partitions which overlap with the query interval are accessed sequentially and distinguished with the help of the auxiliary index. Hence, the contents of the relevant $P^O_{\ell,i}$'s to each query are always accessed sequentially. Figure 4.5(a) shows an example at level $\ell = 4$ of HINT^m. From the total $2^\ell = 16$ P^O partitions at that level, only 5 are nonempty (shown in grey at the top of the figure): $P^O_{4,1}, P^O_{4,5}, P^O_{4,6}, P^O_{4,8}, P^O_{4,13}$. All 9 intervals in them (sorted by start point) are unified in a single table T^O_4 as shown at the bottom of the figure (the binary representations of the interval endpoints are shown). At the moment, ignore the ids column for T^O_4 at the right of the figure. The sparse index for T^O_4 has one entry per nonempty partition pointing to the first interval in it. For the query in the example, the index is used to find the first nonempty partition $P^O_{4,5}$, for which the id is greater than or equal to the 4-bit prefix 0100 of $q.st$. All relevant non-empty partitions $P^O_{4,5}, P^O_{4,6}, P^O_{4,8}$ are accessed sequentially from T^O_4, until the position of the first interval of $P^O_{4,13}$.

Searching for the first partition $P^O_{\ell,f}$ that overlaps with q at each level can be quite expensive when numerous nonempty partitions exist. To alleviate this issue, we suggest adding to the auxiliary index, a link from each partition $P^O_{\ell,i}$ to the partition $P^O_{\ell-1,j}$ at the level above, such that j is the smallest number greater than or equal to $i \div 2$, for which partition $P^O_{\ell-1,j}$ is not empty. Hence, instead of performing binary

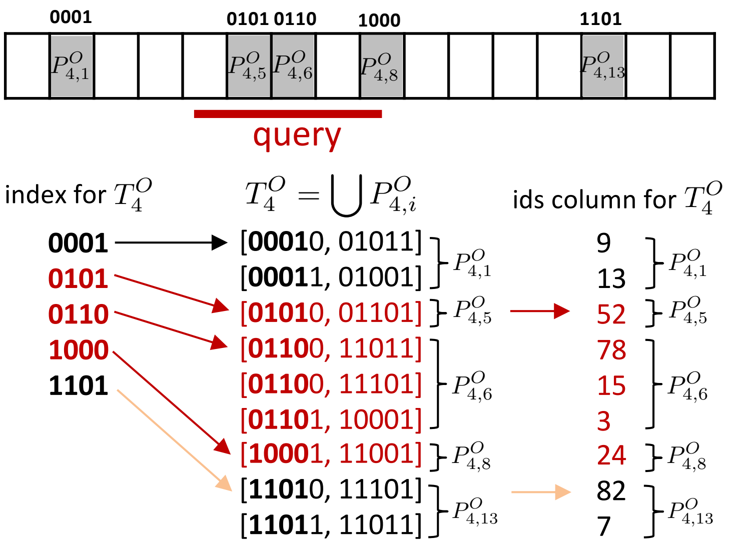
[2]For simplicity, we discuss this organization when a partition $P_{\ell,i}$ is divided into $P^O_{\ell,i}$ and $P^R_{\ell,i}$; the same idea can be straightforwardly applied also when the four subdivisions discussed in Section 4.2.1 are used.

search at level $\ell - 1$, we use the link from the first partition $P^O_{\ell,f}$ relevant to the query at level ℓ and (if necessary) apply a linear search backwards starting from the pointed partition $P^O_{\ell-1,j}$ to identify the first non-empty partition $P^O_{\ell-1,f}$ that overlaps with q. Figure 4.5(b) shows an example, where each nonempty partition at level ℓ is linked with the first nonempty partition with greater than or equal prefix at the level $\ell - 1$ above. Given query example q, we use the auxiliary index to find the first nonempty partition $P^O_{4,5}$ which overlaps with q and also sequentially access $P^O_{4,6}$ and $P^O_{4,8}$. Then, we follow the pointer from $P^O_{4,5}$ to $P^O_{3,4}$ to find the first nonempty partition at level 3, which overlaps with q. We repeat this to get partition $P^O_{2,3}$ at level 2, which however is not guaranteed to be the first one overlapping with q, so we go backwards to $P^O_{2,3}$.

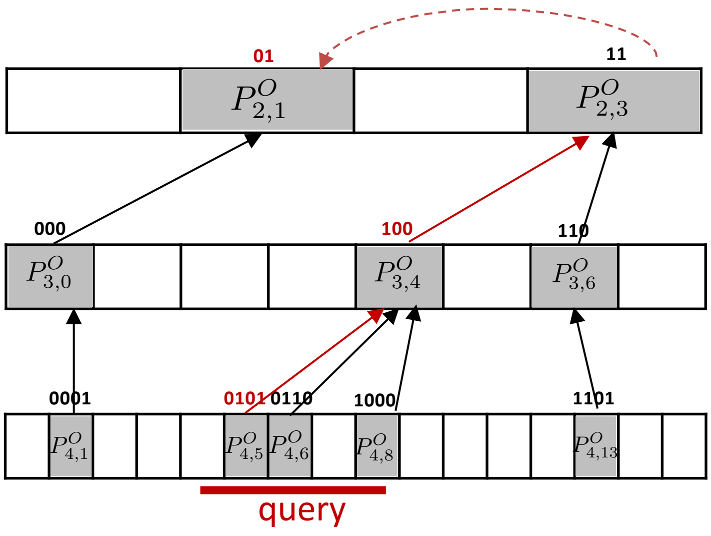
4.2.3 Reducing cache misses

At most levels of HINT^m, no comparisons are conducted and the only operations are processing the interval ids which qualify the query. In addition, even for the levels ℓ where comparisons are required, these are only restricted to the first and the last partitions $P^O_{\ell,f}$ and $P^O_{\ell,l}$ that overlap with q and no comparisons are needed for the partitions that are in-between. Summing up, when accessing any (sub-)partition for which no comparison is required, we do not need any information about the intervals, except for their ids. Hence, in our implementation, for each (sub-)partition, we store the ids of all intervals in it in a dedicated array (the *ids column*) and the interval endpoints (wherever necessary) in a different array.[3] If we need the id of an interval that qualifies a comparison, we can access the corresponding position of the ids column. This storage organization greatly improves search performance by reducing the cache misses, because for the intervals that do not require comparisons, we only access their ids and not their interval endpoints. This optimization is orthogonal to and applied in combination with the strategy in Section 4.2.2, i.e., we store all P^O divisions at each level ℓ in a single table T^O_ℓ, which is decomposed to a column that stores the ids and another table for the endpoint data of the intervals. An example of the ids column is shown in Figure 4.5(a). If, for a sequence of partitions at a level, we do not have to perform any comparisons, we just access the sequence of the interval ids that are part of the answer, which is implied by the position of the first

[3]Similar to the previous section, this storage optimization can be straightforwardly employed also when a partition is divided into $P^{O_{in}}_{\ell,i}$, $P^{O_{aft}}_{\ell,i}$, $P^{R_{in}}_{\ell,i}$, $P^{R_{aft}}_{\ell,i}$.



(a) auxiliary index



(b) linking between levels

Figure 4.5: Storage and indexing optimizations

such partition (obtained via the auxiliary index). In this example, all intervals in $P^O_{4,5}$ and $P^O_{4,6}$ are guaranteed to be query results without any comparisons and they can be sequentially accessed from the ids column without having to access the endpoints of the intervals. The auxiliary index guides the search by identifying and distinguishing between partitions for which comparisons should be conducted (e.g., $P^O_{4,8}$) and those for which they are not necessary.

4.2.4 Updates

A version of HINT^m that uses *all* techniques from Sections 4.2.1-4.2.2, is optimized for query operations. Under this premise, the index cannot efficiently support individual updates, i.e., new intervals inserted one-by-one. Dealing with updates in *batches* will be a better fit. This is a common practice for other update-unfriendly indices, e.g., the inverted index in IR. Yet, for mixed workloads (i.e., with both queries and updates), we adopt a hybrid setting where a *delta* index is maintained to digest the latest updates as discussed in Section 4.1.4,[4] and a fully optimized HINT^m, which is updated periodically in batches, holds older data supporting deletions with tombstones. Both indices are probed when a query is evaluated.

4.3 Experimental Analysis

We compare our hierarchical indexing, detailed in Sections 4.1 and 4.2 against the interval tree [35] (code from [70]), the timeline index [13], the (adaptive) period index [36], and a uniform 1D-grid. All indices were implemented in `C++` and compiled using `gcc` (v4.8.5) with `-O3`. [5] The tests ran on a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz with 384 GBs of RAM, running CentOS Linux.

4.3.1 Data and queries

We used 4 collections of real intervals, which have also been used in previous works; Table 4.3 summarizes their characteristics. BOOKS [27] contains the periods of time in

[4]Small adjustments are applied for the $P^{O_{in}}_{l,i}$, $P^{O_{aft}}_{l,i}$, $P^{R_{in}}_{l,i}$, $P^{R_{aft}}_{l,i}$ subdivisions and the storage optimizations.

[5]Source code available in https://github.com/pbour/hint.

Table 4.3: Characteristics of real datasets

	BOOKS	WEBKIT	TAXIS	GREEND
Cardinality	2,312,602	2,347,346	172,668,003	110,115,441
Size [MBs]	27.8	28.2	2072	1321
Domain [sec]	31,507,200	461,829,284	31,768,287	283,356,410
Min duration [sec]	1	1	1	1
Max duration [sec]	31,406,400	461,815,512	2,148,385	59,468,008
Avg. duration [sec]	2,201,320	33,206,300	758	15
Avg. duration [%]	6.98	7.19	0.0024	0.000005

Table 4.4: Parameters of synthetic datasets

parameter	**values** (defaults in **bold**)
Domain length	32M, 64M,**128M**, 256M, 512M
Cardinality	**10M**, 50M, 100M, 500M, 1B
α (interval length)	1.01, 1.1, **1.2**, 1.4, 1.8
σ (interval position)	10K, 100K, **1M**, 5M, 10M

2013 when books were lent out by Aarhus libraries (https://www.odaa.dk). WEBKIT [27, 49, 25, 26] records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); the intervals indicate the periods during which a file did not change. TAXIS [28] stores the time periods of taxi trips (pick-up and drop-off timestamps) from NY City in 2013 (https://www1.nyc.gov/site/tlc/index.page). GREEND [29, 71] records time periods of power usage from households in Austria and Italy from January 2010 to October 2014. BOOKS and WEBKIT contain around 2M intervals each, which are quite long on average; TAXIS and GREEND have over 100M short intervals.

We also generated synthetic collections to simulate different cases for the lengths and the skewness of the input intervals. Table 4.4 shows the construction parameters for the synthetic datasets and their default values. The domain of the datasets ranges from 32M to 512M, which requires index level parameter m to range from 25 to 29 for a comparison-free HINT (similar to the real datasets). The cardinality ranges from 10M to 1B. The lengths of the intervals were generated using the `random.zipf(`α`)` function in the `numpy` library. They follow a zipfian distribution according to the $p(x) = \frac{x^{-a}}{\zeta(a)}$ probability density function, where ζ is the Riemann Zeta function. A small value of α results in most intervals being relatively long, while a large value results in the great majority of intervals having length 1. The positions of the *middle points* of the intervals are generated from a normal distribution centered at the middle

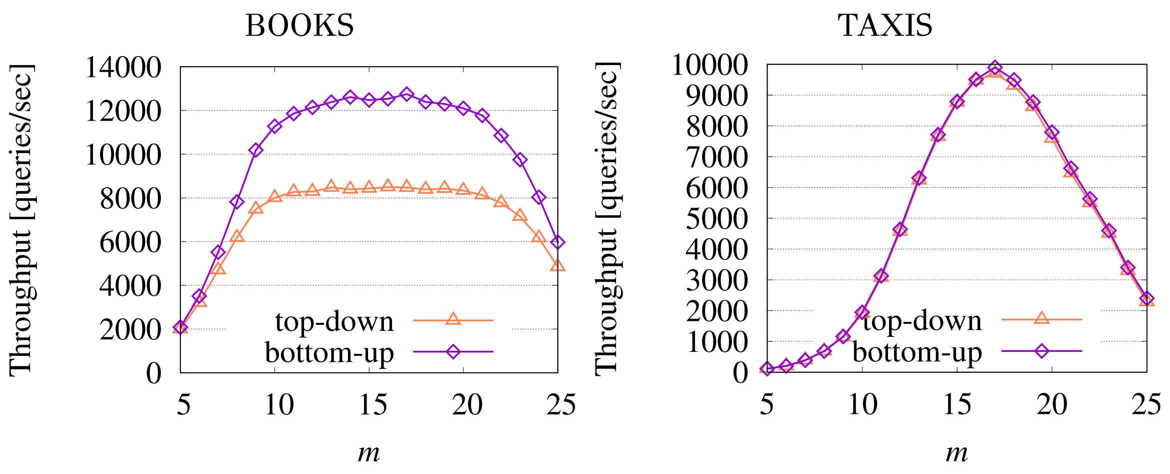


Figure 4.6: Optimizing HINT^m: query evaluation approaches

point μ of the domain. Hence, the middle point of each interval is generated by calling numpy's `random.normalvariate(`μ, σ`)`. The greater the value of σ the more spread the intervals are in the domain.

On the real datasets, we applied queries uniformly distributed in the domain. On the synthetic, the query positions follow the distribution of the data. In both cases, the extent of the query intervals were fixed to a percentage of the domain size (default 0.1%). At each test, we ran 10K random queries, in order to measure the overall throughput. Measuring query throughput instead of average time per query makes sense in applications or services that manage huge volumes of interval data and offer a search interface to billions of users simultaneously (e.g., public historical databases).

4.3.2 Optimizing HINT/HINT^m

In our first set of experiments, we study the best setting for our hierarchical indexing. Specifically, we compare the effectiveness of the two query evaluation approaches discussed in Section 4.1.2 and investigate the impact of the optimizations described in Section 4.2.

Query evaluation approaches on HINT^m

We compare the straightforward *top-down* approach for evaluating queries on HINT^m that uses solely Lemma 4.1, against the *bottom-up* illustrated in Algorithm 4.3 which additionally employs Lemma 4.2. Figure 4.6 reports the throughput of each approach on BOOKS and TAXIS, while varying the number of levels m in the index. Due to lack of space, we omit the results for WEBKIT and GREEND that follow exactly the same trend with BOOKS and TAXIS, respectively. We observe that the *bottom-up* approach significantly outperforms *top-down* for BOOKS while for TAXIS, this performance gap is very small. As expected, *bottom-up* performs at its best for inputs that contain long intervals which are indexed on high levels of the index, i.e., the intervals in BOOKS. In contrast, the intervals in TAXIS are very short and so, indexed at the bottom level of HINT^m, while the majority of the partitions at the higher levels are empty. As a result, *top-down* conducts no comparisons at higher levels. For the rest of our tests, HINT^m uses the *bottom-up* approach (i.e., Algorithm 4.3).

Subdivisions and space decomposition

We next evaluate the *subdivisions* and *space decomposition* optimizations described in Section 4.2.1 for HINT^m. Note that these techniques are not applicable to our comparison-free HINT as the index stores only interval ids. Figure 4.7 shows the effect of the optimizations on BOOKS and TAXIS, for different values of m; similar trends were observed in WEBKIT and GREEND, respectively. The plots include (1) a *base* version of HINT^m, which employs none of the proposed optimizations, (2) *subs+sort+opt*, with all optimizations activated, (3) *subs+sort*, which only sorts the subdivisions (section 4.2.1) and (iv) *subs+sopt*, which uses only the storage optimization for the subdivisions (Section 4.2.1). We observe that the *subs+sort+opt* version of HINT^m is superior to all three other versions, on all tests. Essentially, the index benefits from the *sub+sort* setting only when m is small, i.e., below 15, at the expense of increasing the index time compared to *base*. In this case, the partitions contain a large number of intervals and therefore, using binary search or scanning until the first interval that does not overlap the query, will save on the conducted comparisons. On the other hand, the *subs+sopt* optimization significantly reduces the space requirements of the index. As a result, the version incurs a higher cache hit ratio and so, a higher throughput compared to *base* is achieved, especially for large values of m, i.e., higher than 10. The *subs+sort+opt* version manages to combine the benefits of both *subs+sort* and *subs+sopt* versions, i.e., high throughput in all cases, with low space requirements. The effect in the performance is more pronounced in BOOKS because of the long intervals and the high replication ratio. In view of these results, HINT^m employs all optimizations from Section 4.2.1 for the rest of our experiments.

Table 4.5: Optimizing HINT: impact of the skewness & sparsity optimization (Section 4.2.2), default parameters

dataset	**throughput** [queries/sec]		**index size** [MBs]	
	original	optimized	original	optimized
BOOKS	12098	36173	3282	273
WEBKIT	947	39000	49439	337
TAXIS	2931	31027	10093	7733
GREEND	648	47038	57667	10131

Handling data skewness & sparsity and reducing cache misses

Table 4.5 tests the effect of the *handling data skewness & sparsity* optimization (Section 4.2.2) on the comparison-free version of HINT (Section 4.1.1).[6] Observe that the optimization has a great effect on both the throughput and the size of the index in all four real datasets, because empty partitions are effectively excluded from query evaluation and from the indexing process.

Figure 4.8 shows the effect of either or both of the *data skewness & sparsity* (Section 4.2.2) and the *cache misses* optimizations (Section 4.2.3) on the performance of HINT^m for different values of m. In all cases, the version of HINT^m which uses both optimizations is superior to all other versions. As expected, the *skewness & sparsity* optimization helps to reduce the space requirements of the index when m is large, because there are many empty partitions in this case at the bottom levels of the index. At the same time, the *cache misses* optimization helps in reducing the number of cache misses in all cases where no comparisons are needed. Overall, the optimized version of $HINT^m$ converges to its best performance at a relatively small value of m, where the space requirements of the index are relatively low, especially on the BOOKS and WEBKIT datasets which contain long intervals. For the rest of our experiments, HINT^m employs both optimizations and HINT the *data skewness & sparsity* optimization.

Tuning m

After demonstrating the merit of HINT^m optimizations, we now elaborate on how to set the value of m and on the effectiveness of our analytical model from Section 4.1.3. As we already discussed our model is based on the intuition that as m increases, the cost of accessing comparison-free results dominates the computational cost of the comparisons. Figure 4.9 confirms our intuition on BOOKS and TAXIS (the plots for WEBKIT and GREEND exhibit exactly the same trend as BOOKS and TAXIS, respectively). For different values of m and for $10K$ queries, we report the overall time spend for comparisons between data intervals and query intervals, denoted by C_{cmp} , and the overall time spent to output results with no comparisons, denoted by C_{acc}, i.e., the time taken for simply accessing data intervals which are guaranteed query results. We also include the total execution time, i.e., $C_{cmp} + C_{acc}$.

The plots clearly show the expected behaviour. For small values of m, the cost of conducting comparisons dominates the total execution cost, because the partitions at the bottom level m of the index have large extents and numerous intervals. As the value of m increases, the fraction of the results collected from just accessing the contents of partitions rises, increasing the C_{acc} cost. The optimal values m_{opt} (i.e., where the total execution time is the lowest possible occur after C_{acc} exceeds C_{cmp}. In fact, we notice that increasing m beyond m_{opt} roughly eliminates the cost of comparisons ($C_{cmp} \approx 0$), because the partitions are much shorter than the query intervals, while the total cost essentially equals the cost of simply accessing the intervals from the comparison-free partitions.

To determine m_{opt}, our model in Section 4.1.3 selects the smallest m value for which the index converges within 3% to its lowest estimated cost. Table 4.6 reports, for each real dataset, m_{opt} (est.) and (2) m_{opt} (exps), which brings the highest throughput in our tests. Overall, our model estimates a value of m_{opt} which is very close to the experimentally best value of m. Despite a larger gap for WEBKIT, the measured throughput for the estimated $m_{opt} = 9$ is only 5% lower than the best observed throughput.

Table 4.6: Statistics and parameter setting

index	parameter	BOOKS	WEBKIT	TAXIS	GREEND
Period	#levels	4	4	7	8
	#coarse partitions	100	100	100	100
Timeline	#checkpoints	6000	6000	8000	8000
1D-grid	#partitions	500	300	4000	30000
HINT^m	m_{opt} (est.)	9	9	16	16
	m_{opt} (exps)	10	12	17	17
	rep. factor k (est.)	6.09	8.98	1.98	1
	rep. factor k (exps)	5.13	6.07	2.14	1.0013
	avg. comp. part.	3.226	3.538	3.856	2.937
	no comp. results	99.9%	99.9%	99.8%	99.3%

Discussion

Table 4.6 also shows the *replication factor k* of the index, i.e., the average number of partitions in which every interval is stored, as predicted by our space complexity analysis (see Theorem 4.1) and as measured experimentally. As expected, the replication factor is high on BOOKS, WEBKIT due to the large number of long intervals, and low on TAXIS, GREEND where the intervals are very short and stored at the bottom levels. Although our analysis uses simple statistics, the predictions are quite accurate.

The next line of the table (*avg. comp. part.*) shows the average number of HINT^m partitions for which comparisons were conducted. Consistently to our analysis in Section 4.1.2, all numbers are below 4, which means that the performance of HINT^m is very close to the performance of the comparison-free, but space-demanding HINT. To further elaborate on the number of required comparisons, we last show the fraction of the results produced by HINT^m without any comparisons. We observe that in all datasets over 99% of the results are collected with no comparisons, which further explains how HINT^m is able to match the performance of the comparison-free HINT.

4.3.3 Index performance comparison

Next, we compare the optimized versions of HINT and HINT^m against the previous work competitors. We start with our tests on the real datasets. For HINT^m, we set m to the best value on each dataset, according to Table 4.6. Similarly, we set the number of partitions for 1D-grid, the number of checkpoints for the timeline index,

[6]The *cache misses* optimization (Section 4.2.3) is only applicable to HINT^m.

Table 4.7: Comparing index size [MBs]

index	BOOKS	WEBKIT	TAXIS	GREEND
Interval tree	97	115	3125	2241
Period	210	217	2278	**1262**
Timeline	4916	5671	4203	2525
1D-grid	949	604	2165	1264
HINT	273	337	7733	10131
HINT^m	**81**	**98**	**2039**	1278

Table 4.8: Comparing index time [sec]

index	BOOKS	WEBKIT	TAXIS	GREEND
Interval tree	**0.25**	**0.33**	47.2	26.8
Period	1.15	1.35	76.9	46.4
Timeline	12.7	19.2	40.4	15.9
1D-grid	1.26	0.95	**4.02**	**2.24**
HINT	1.70	11.8	49.6	36.5
HINT^m	0.73	0.53	22.8	8.58

and the number of levels and number of coarse partitions for the period index (see Table 4.6). Table 4.7 shows the sizes of each index in memory and Table 4.8 shows the construction cost of each index, for the default query extent 0.1%. Regarding space, HINT^m along with the interval tree and the period index have the lowest requirements on datasets with long intervals (BOOKS and WEBKIT) and very similar to 1D-grid in the rest. In TAXIS and GREEND where the intervals are indexed mainly at the bottom level, the space requirements of HINT^m are significantly lower than our comparison-free HINT due to limiting the number of levels. When compared to the raw data (see Table 4.3), HINT^m is 2 to 3 times bigger for BOOKS and WEBKIT (which contain many long intervals), and 1 time bigger for GREEND and TAXIS. These ratios are smaller than the replication ratios k reported in Table 4.6, thanks to our storage optimization (cf. Section 4.2.1). Due to its simplicity, 1D-grid has the lowest index time across all datasets. Nevertheless, HINT^m is the runner up in most of the cases, especially for the biggest inputs, i.e., TAXIS and GREEND, while in BOOKS and WEBKIT, its index time is very close to the interval tree.

Figure 4.10 compares the query throughputs of all indices on queries of various extents (as a percentage of the domain size). The first set of bars in each plot corresponds to *stabbing* queries, i.e., queries of 0 extent. We observe that HINT and HINT^m outperform the competition by almost one order of magnitude, across the board. In fact, only on GREEND the performance for one of the competitors, i.e., 1D-grid, comes close to the performance of our hierarchical indexing. Due to the extremely short intervals in GREEND (see Table 4.3) the vast majority of the results are collected from the bottom level of HINT/HINT^m, which essentially resembles the evaluation process in 1D-grid. Nevertheless, our indices are even in this case faster as they require no duplicate elimination.

HINT^m is the best index overall, as it achieves the performance of HINT, requiring less space, confirming the findings of our analysis in Section 4.1.2. As shown in Table 4.7, HINT always has higher space requirements than HINT^m; even up to an order of magnitude higher in case of GREEND. What is more, since HINT^m offers the option to control the occupied space in memory by appropriately setting the m parameter, it can handle scenarios with space limitations. HINT is marginally better than HINT^m only on datasets with short intervals (TAXIS and GREEND) and only for selective queries. In these cases, the intervals are stored at the lowest levels of the hierarchy where HINT^m typically needs to conduct comparisons to identify results, but HINT applies comparison-free retrieval.

The next set of tests are on synthetic datasets. In each test, we fix all but one parameters (domain size, cardinality, α, σ, query extent) to their default values and varied one (see Table 4.4). The value of m for HINT^m, the number of partitions for 1D-grid, the number of checkpoints for the timeline index and the number of levels/coarse partitions for the period index are set to their best values on each dataset. The results, shown in Figure 4.11, follow a similar trend to the tests on the real datasets. HINT and HINT^m are always significantly faster than the competition, . Different to the real datasets, 1D-grid is steadily outperformed by the other three competitors. Intuitively, the uniform partitioning of the domain in 1D-grid cannot cope with the skewness of the synthetic datasets. As expected the domain size, the dataset cardinality and the query extent have a negative impact on the performance of all indices. Essentially, increasing the domain size under a fixed query extent, affects the performance similar to increasing the query extent, i.e., the queries become longer and less selective, including more results. Further, the querying cost grows linearly with the dataset size since the number of query results are proportional to it. HINT^m occupies around 8% more space than the raw data, because the replication factor k is close to 1. In contrast, as α grows, the intervals become shorter, so the query performance improves. Similarly, when increasing σ the intervals are more widespread, meaning that the queries are expected to retrieve fewer results, and the query cost drops accordingly.

Table 4.9: Throughput [operations/sec], total cost [sec]

BOOKS

index	**operation**			**total cost**
	queries	insertions	deletions	
Interval tree	1,258	5,841	1,142	9.63
Period index	3,088	519,904	765	4.52
1D-grid	3,739	411,540	165	8.68
$_{\text{subs+sopt}}$HINT^m	14,390	2,405,228	2,201	1.14
HINT^m	**40,311**	**3,680,457**	**5,928**	**0.41**

TAXIS

index	**operation**			**total cost**
	queries	insertions	deletions	
Interval tree	2,619	61,923	14,318	3.93
Period index	2,695	1,026,423	21,293	3.76
1D-grid	2,572	**8,347,273**	16,236	3.95
$_{\text{subs+sopt}}$HINT^m	8,774	4,407,743	71,122	71,122
HINT^m	**28,596**	6,745,622	**90,460**	**0.36**

4.3.4 Updates

We now test the efficiency of HINT^m in updates using both the update-friendly version of HINT^m (Section 4.1.4), denoted by $_{\text{subs+sopt}}$HINT^m, and the hybrid setting for the fully-optimized index from Section 4.2.4, denoted as HINT^m. We index offline the first 90% of the intervals for each real dataset in batch and then execute a mixed workload with 10K queries of 0.1% extent, 5K insertions of new intervals (randomly selected from the remaining 10% of the dataset) and 1K random deletions. Table 4.9 reports our findings for BOOKS and TAXIS; the results for WEBKIT and GREEND follow the same trend. Note that we excluded Timeline since the index is designed for temporal (versioned) data where updates only happen as new events are appended at the end of the event list, and the comparison-free HINT, for which our tests have already shown a similar performance to HINT^m with higher indexing/ storing costs. Also, all indices handle deletions with "tombstones". We observe that both versions of HINT^m outperform the competition by a wide margin. An exception arises on TAXIS, as the short intervals are inserted in only one partition in 1D-grid. The interval tree has in fact several orders of magnitude slower updates due to the extra cost of maintaining the partitions in the tree sorted at all time. Overall, we also observe that the hybrid HINT^m setting is the most efficient index as the *smaller* delta $_{\text{subs+sopt}}$HINT^m handles insertions faster than the 90% pre-filled $_{\text{subs+sopt}}$HINT^m.

4.3.5 Interval Joins

The last experiment in the first part of our analysis investigates the applicability of HINT^m to the evaluation of interval joins. In this operation, given two input datasets R, S, the objective is to find all pairs of intervals $(r, s), r \in R, s \in S$, such that r G-OVERLAPS with s.

When none of the R, S input collections are indexed, we employ the optFS algorithm from [27, 28] to compute the $R \bowtie S$ join. In what follows, we discuss how to further enhance the performance of this approach by extending optFS towards two directions. First, the study in [27, 28] considered a space-based partitioning (referred to as "domain-based partitioning") solely as a means for processing the join in parallel. Here, we extend optFS to employ such a partitioning also for the single-threaded computation. Second, a domain partition in [27, 28] from each input collection is split into 3 sub-divisions; P^O, $P^{R_{in}}$ and $P^{R_{aft}}$ (referred to as classes A, B and C, respectively in [27, 28]). Sub-divisions $P^{R_{in}}$ and $P^{R_{aft}}$ are identical to the ones considered by HINT, while P^O consists of all intervals starting inside a partition, regardless where they end, i.e., $P^O = P^O_{in} \bigcup P^O_{aft}$. We extend the domain-based partitioning to consider all 4 sub-divisions introduced in Chapter 4.

Specifically, the unified $\mathcal{R} \bigcup S$ domain is first split into k equally sized, non-overlapping stripes; each stripe holds a partition for input R and one for S. An interval $s \in S$ (resp. $r \in R$) is assigned to the partition of the stripe that contains $s.st$ and replicated to the partitions of all other stripes it intersects. With this partitioning in place an $R \bowtie S$ join is broken down into k independent partition-to-partition joins, i.e., $R : P_1 \bowtie S : P_1$, ..., $R : P_k \bowtie S : P_k$. Next, in a similar fashion to HINT, every domain partition P from each collection, is further divided into 4 sub-divisions $P^{O_{in}}$, $P^{O_{aft}}$, $P^{R_{in}}$ and $P^{R_{aft}}$. With these subdivisions in place, we now break down every partition-to-partition join into 16 smaller tasks, called *mini-joins*. Figure 4.12 illustrates this mini-joins breakdown. We next elaborate on the computation of every mini-join type:

- The 3 original-to-original mini-joins $R : P^{O_{in}} \bowtie S : P^{O_{in}}$, $R : P^{O_{in}} \bowtie S : P^{O_{aft}}$ and $R : P^{O_{aft}} \bowtie S : P^{O_{in}}$ have identical complexity to the original $R \bowtie S$ join. Therefore, these mini-joins are evaluated as normal, i.e., using the optimized FS algorithm, optFS from [27, 28].

- The remaining original-to-original mini-join $R : P^{O_{aft}} \bowtie S : P^{O_{aft}}$ differs from the previous case. By definition, a (r, s) pair of intervals in this case always overlap, regardless where their start is located inside the corresponding stripe, as they both span to the next stripe. Under this premise, $R : P^{O_{aft}} \bowtie S : P^{O_{aft}}$ is computed without conducting any comparisons, in a cross-product fashion; we highlight this mini-join in pink color, in Figure 4.12.

- For the 4 original-to-replica mini-joins $R : P^{O_{in}} \bowtie S : P^{R_{in}}$, $R : P^{O_{aft}} \bowtie S : P^{R_{in}}$, $R : P^{R_{in}} \bowtie S : P^{O_{in}}$ and $R : P^{R_{in}} \bowtie S : P^{O_{aft}}$, a simplified (reduced) version of optFS can be used. As every replica interval inside $P^{R_{in}}$ starts in a preceding stripe, optFS only conducts forward scans to the $P^{O_{in}}$ or $P^{O_{aft}}$ sub-divisions from the other input; no forward scans are needed for the original intervals. In addition, if the grouping optimization is activated, the entire $P^{R_{in}}$ is used as a group. We highlight this mini-join type in blue, in Figure 4.12.

- Every interval in $P^{R_{aft}}$ spans the entire range of the corresponding domain stripe. Thus, such intervals intersect by definition with all intervals from the other input that start inside the same stripe, i.e., the intervals stored inside the $P^{O_{in}}$ and $P^{O_{aft}}$ sub-divisions. Hence, the 4 original-to-replicas mini-joins $R : P^{O_{in}} \bowtie S : P^{R_{aft}}$, $R : P^{O_{aft}} \bowtie S : P^{R_{aft}}$, $R : P^{R_{aft}} \bowtie S : P^{O_{in}}$ and $R : P^{R_{aft}} \bowtie S : P^{O_{aft}}$ are computed as cross-products and we color them in pink in Figure 4.12.

- Last, to avoid producing duplicate results, a join result (r, s) is reported only if at least one of the involved intervals is not a replica, i.e., if it is not contained inside a $P^{R_{in}}$ or a $P^{R_{aft}}$ sub-division. Under this premise, we never compute the 4 replica-to-replica mini-joins $R : P^{R_{in}} \bowtie S : P^{R_{in}}$, $R : P^{R_{in}} \bowtie S : P^{R_{aft}}$, $R : P^{R_{aft}} \bowtie S : P^{R_{in}}$ and $R : P^{R_{aft}} \bowtie S : P^{R_{aft}}$, shaded in orange color in Figure 4.12.

The rationale is that if the outer dataset R is very small compared to the inner dataset S, an index already available for S can be used to evaluate fast the join in an index nested loops fashion. Hence, we show how HINT^m constructed for each of the four real datasets can be used to evaluate joins where the outer relation is a random sample of the same dataset. As part of the join process, we sort the outer dataset R in order to achieve better cache locality between consecutive probes to the inner dataset S. As a competitor, we used the state-of-the-art interval join algorithm [28], which sorts both join inputs and applies a specialized sweeping algorithm optFS. Figure 4.15 shows the results for various sizes $|R|$ of the outer dataset R. The results confirm our expectation. For small sizes of $|R|$, HINT^m is able to outperform the algorithm of [28]. On the TAXIS dataset, in particular, HINT^m loses to [28] only when $|R|/|S| \geq 50\%$.

4.4 Conclusions

We proposed a hierarchical index (HINT) for intervals, which has low space complexity and minimizes the number of data accesses and comparisons during query evaluation. Our experiments on real and synthetic datasets shows that HINT outperforms previous work by one order of magnitude in a wide variety of interval data and query distributions.

Algorithm 4.3 Searching HINT^m

```
Input   : HINT$^m$ index $\mathcal{H}$, query interval $q$
Output  : set $\mathcal{R}$ of intervals that overlap with $q$
$compfirst \leftarrow TRUE$; $complast \leftarrow TRUE$
$\mathcal{R} \leftarrow \emptyset$
for $\ell = m$ to $0$ do                                                  ▷ bottom-up
    $f \leftarrow prefix(\ell, q.st)$; $l \leftarrow prefix(\ell, q.end)$
    for $i = f$ to $l$ do
        if $i = f$ then                                                   ▷ first overlapping partition
            if $i = l$ and $compfirst$ and $complast$ then
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i}, q.st \leq s.end \wedge s.st \leq q.end\}$
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^R_{\ell,i}, q.st \leq s.end\}$
            end
            else if $i = l$ and $complast$ then
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i}, s.st \leq q.end\}$
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^R\}$
            end
            else if $compfirst$ then
                $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i} \cup \mathcal{H}.P^R_{\ell,i}, q.st \leq s.end\}$
            end
            else
                $\mathcal{R} \leftarrow$ ‘$\mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i} \cup \mathcal{H}.P^R_{\ell,i}\}$
            end
        end
        else if $i = l$ and $complast$ then                               ▷ last partition, $l > f$
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i}, s.st \leq q.end\}$
        end
        else                                          ▷ in-between or last ($l > f$), no comparisons
            $\mathcal{R} \leftarrow \mathcal{R} \cup \{s.id | s \in \mathcal{H}.P^O_{\ell,i}\}$
        end
    end
    if $f$ $mod$ $2 = 0$ then                                              ▷ last bit of $f$ is 0
        $compfirst \leftarrow FALSE$
    end
    if $l$ $mod$ $2 = 1$ then                                              ▷ last bit of $l$ is 1
        $complast \leftarrow FALSE$
    end
end
return $\mathcal{R}$
```

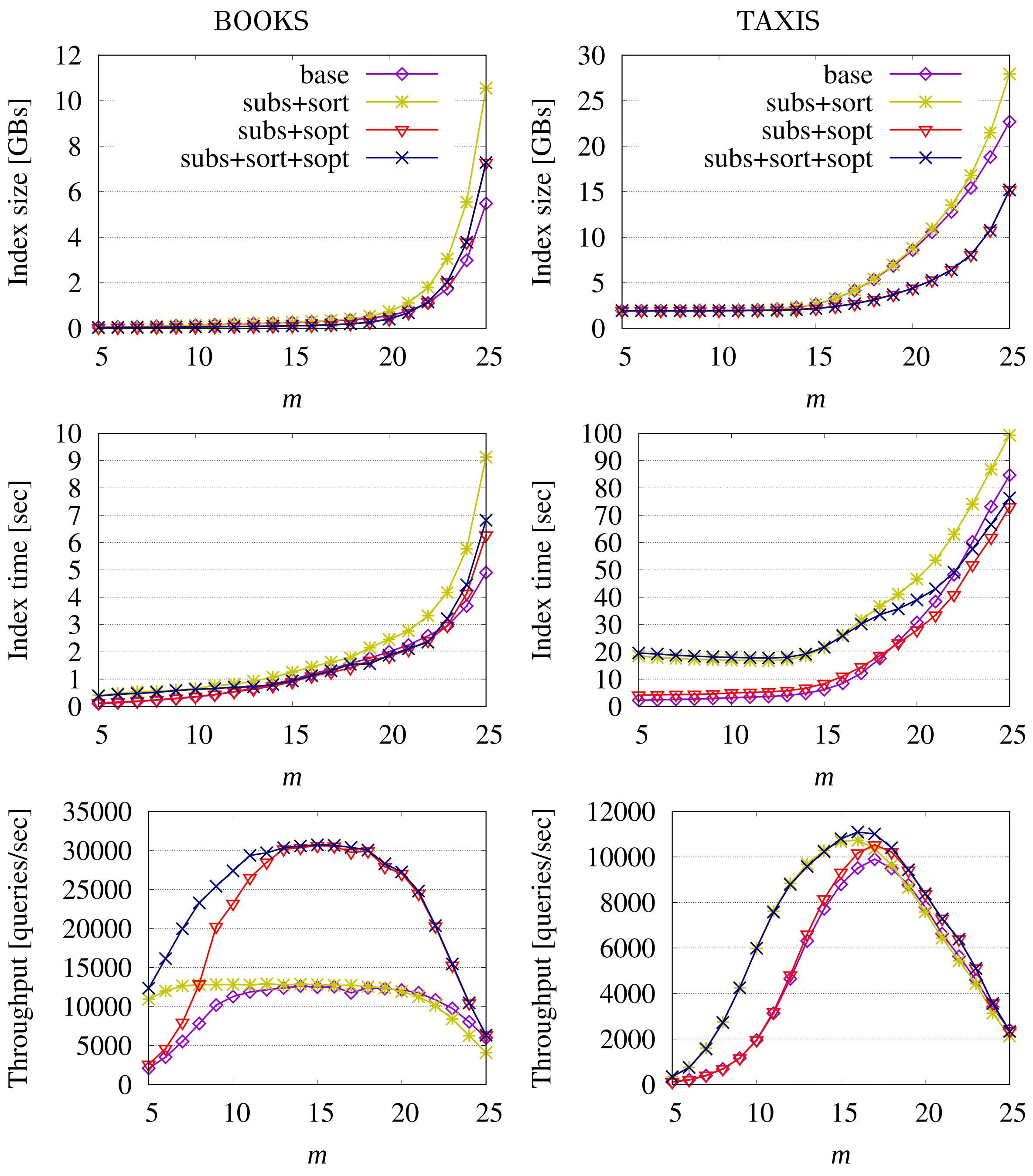


Figure 4.7: Optimizing HINT^m: subdivisions and space decomposition

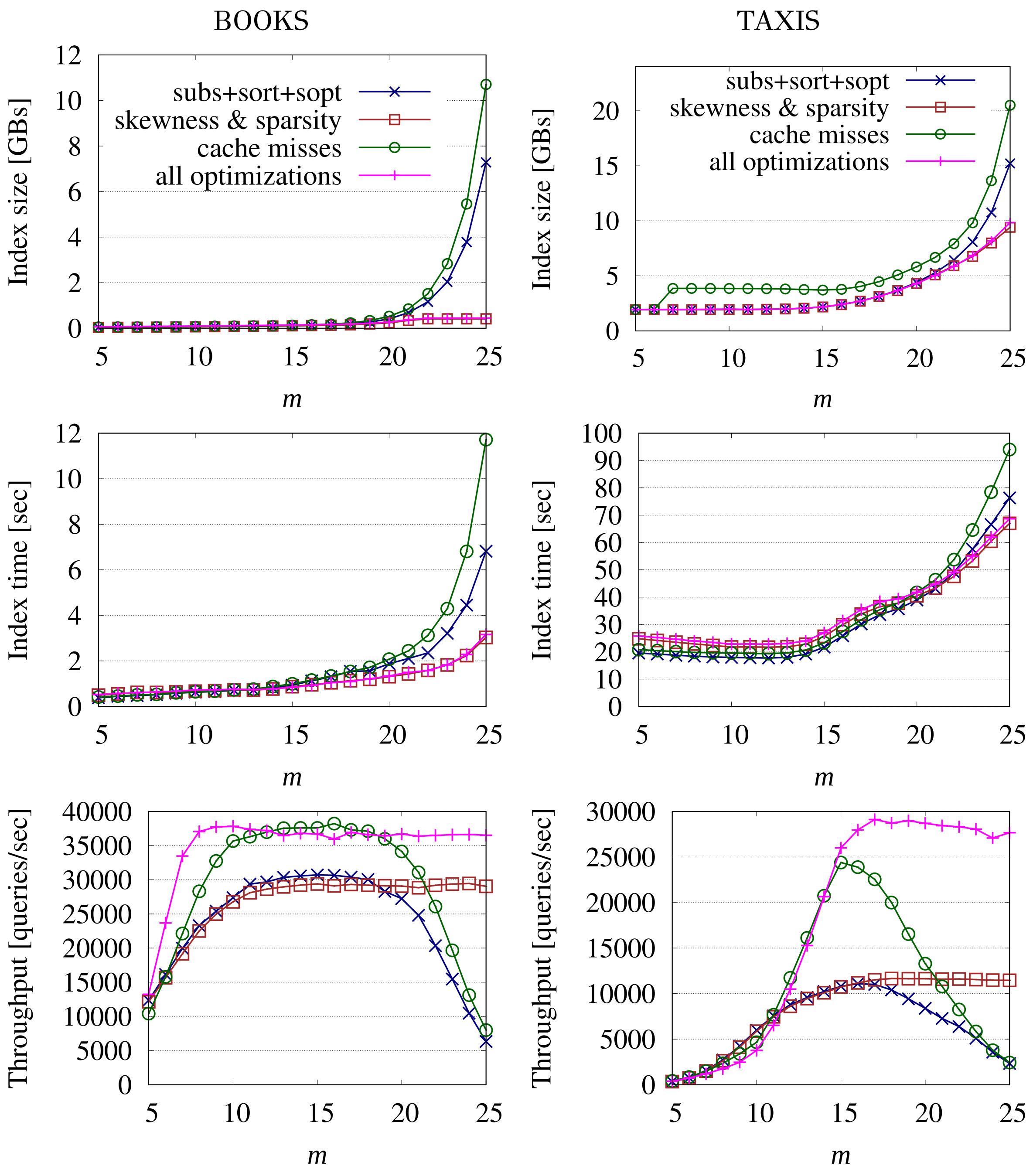


Figure 4.8: Optimizing HINT^m: impact of handling skewness & sparsity and reducing cache misses optimizations

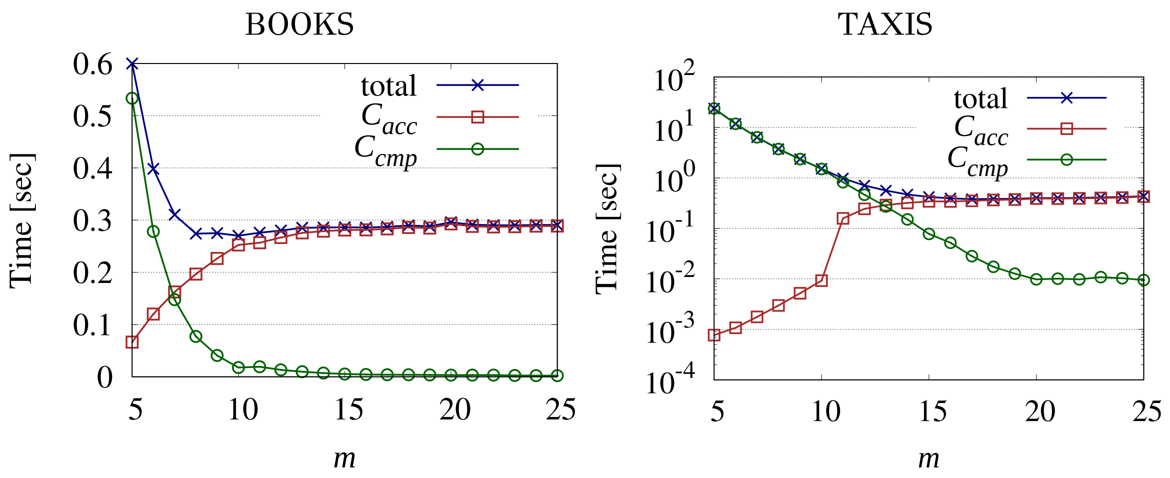


Figure 4.9: Setting m: measured costs

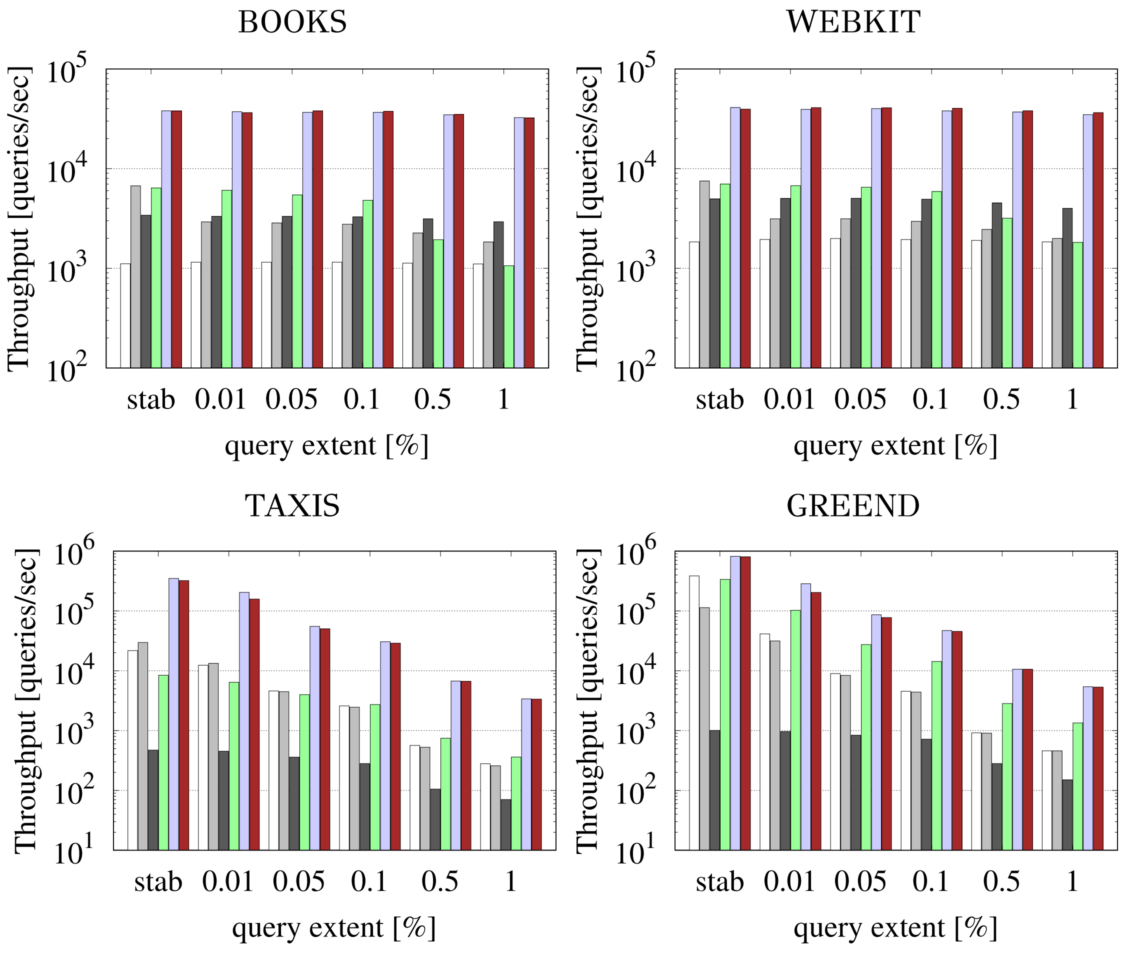


Figure 4.10: Comparing throughputs, real datasets

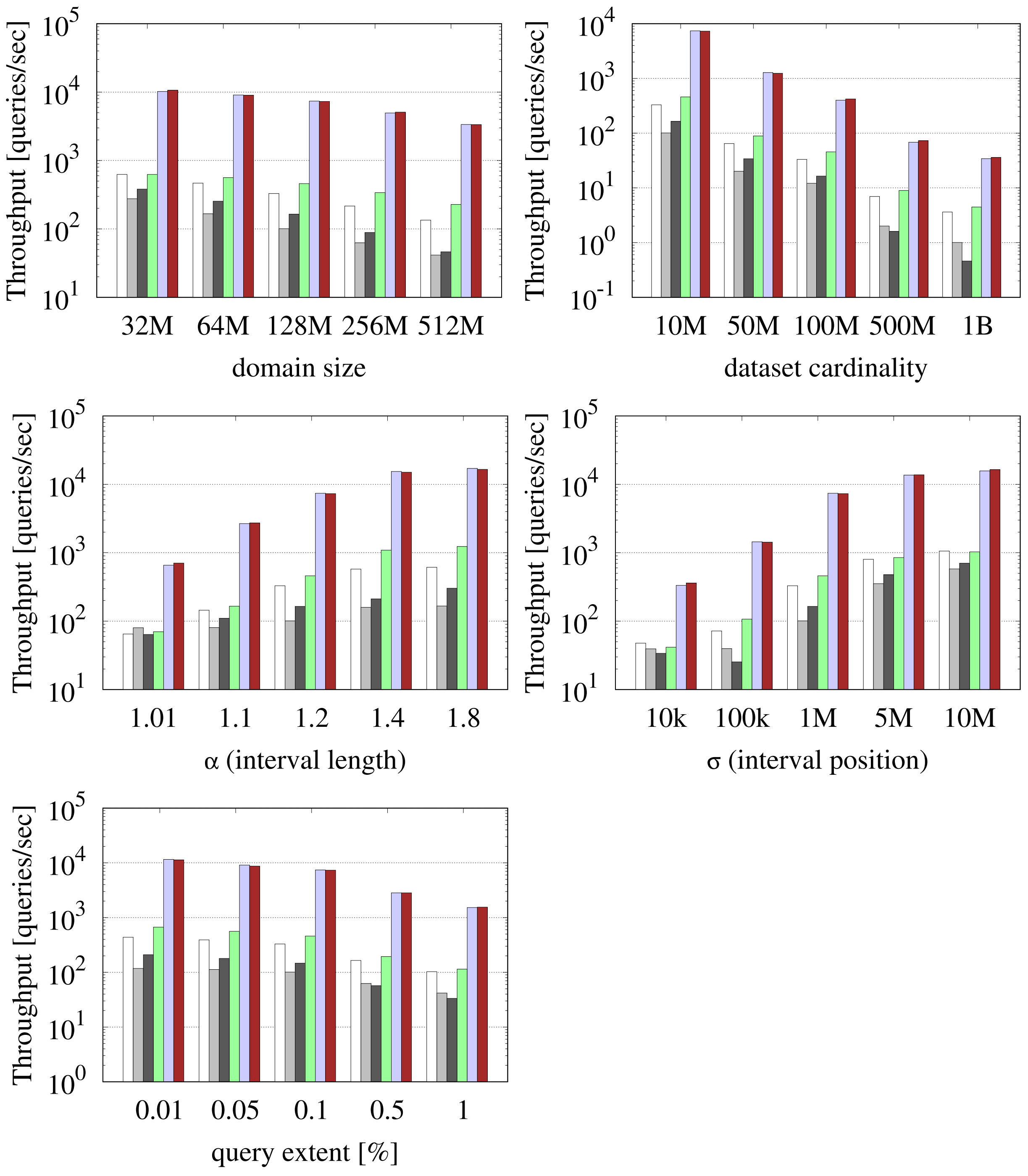


Figure 4.11: Comparing throughputs, synthetic datasets

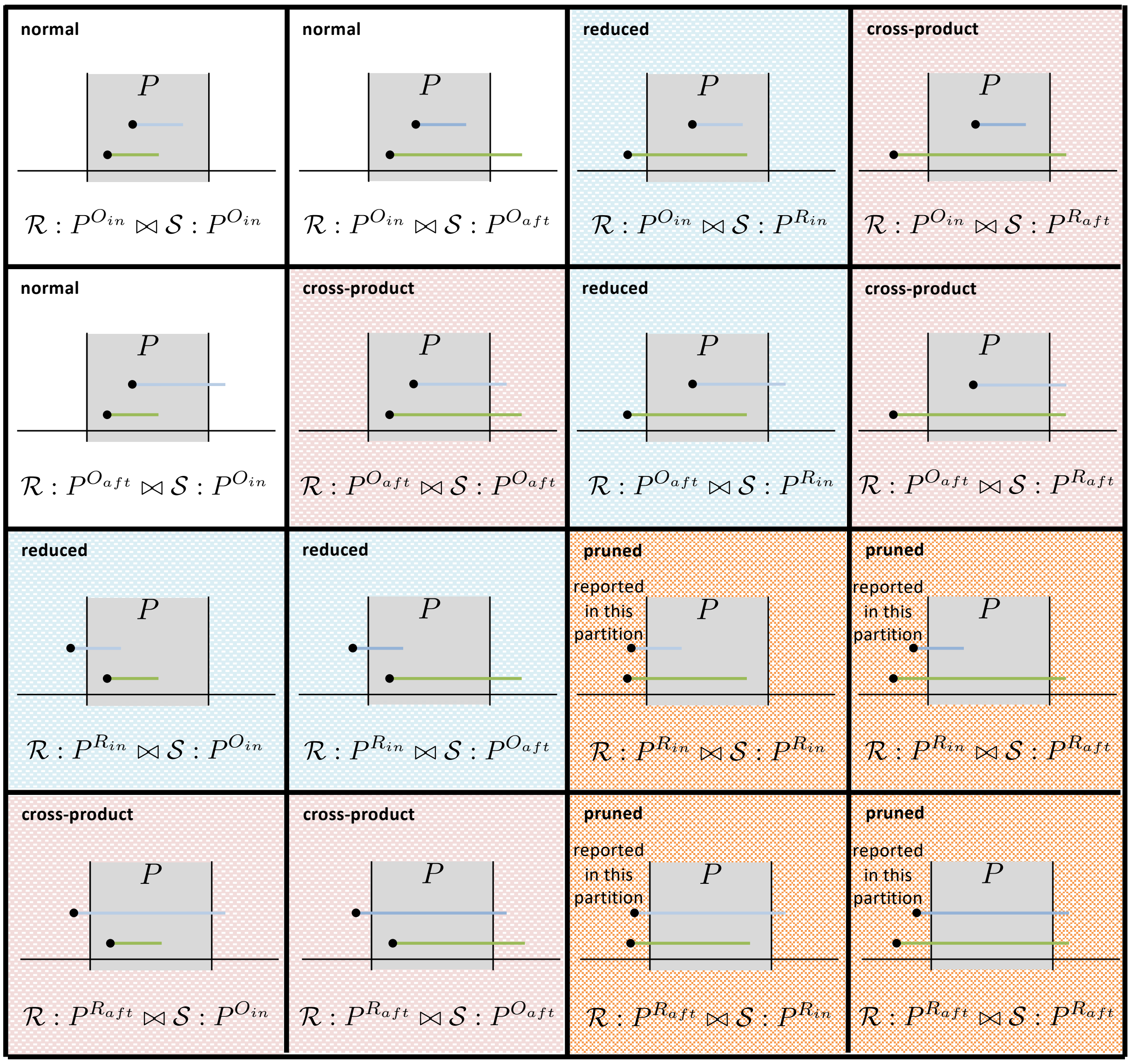

normal
P
$\mathcal{R}: P^{O_{in}} \bowtie \mathcal{S}: P^{O_{in}}$
normal
P
$\mathcal{R}: P^{O_{in}} \bowtie \mathcal{S}: P^{O_{aft}}$
reduced
P
$\mathcal{R}: P^{O_{in}} \bowtie \mathcal{S}: P^{R_{in}}$
cross-product
P
$\mathcal{R}: P^{O_{in}} \bowtie \mathcal{S}: P^{R_{aft}}$
normal
P
$\mathcal{R}: P^{O_{aft}} \bowtie \mathcal{S}: P^{O_{in}}$
cross-product
P
$\mathcal{R}: P^{O_{aft}} \bowtie \mathcal{S}: P^{O_{aft}}$
reduced
P
$\mathcal{R}: P^{O_{aft}} \bowtie \mathcal{S}: P^{R_{in}}$
cross-product
P
$\mathcal{R}: P^{O_{aft}} \bowtie \mathcal{S}: P^{R_{aft}}$
reduced
P
$\mathcal{R}: P^{R_{in}} \bowtie \mathcal{S}: P^{O_{in}}$
reduced
P
$\mathcal{R}: P^{R_{in}} \bowtie \mathcal{S}: P^{O_{aft}}$
pruned
reported in this partition
P
$\mathcal{R}: P^{R_{in}} \bowtie \mathcal{S}: P^{R_{in}}$
pruned
reported in this partition
P
$\mathcal{R}: P^{R_{in}} \bowtie \mathcal{S}: P^{R_{aft}}$
cross-product
P
$\mathcal{R}: P^{R_{aft}} \bowtie \mathcal{S}: P^{O_{in}}$
cross-product
P
$\mathcal{R}: P^{R_{aft}} \bowtie \mathcal{S}: P^{O_{aft}}$
pruned
reported in this partition
P
$\mathcal{R}: P^{R_{aft}} \bowtie \mathcal{S}: P^{R_{in}}$
pruned
reported in this partition
P
$\mathcal{R}: P^{R_{aft}} \bowtie \mathcal{S}: P^{R_{aft}}$

Figure 4.12: Mini-joins breakdown for partition-to-partition joins

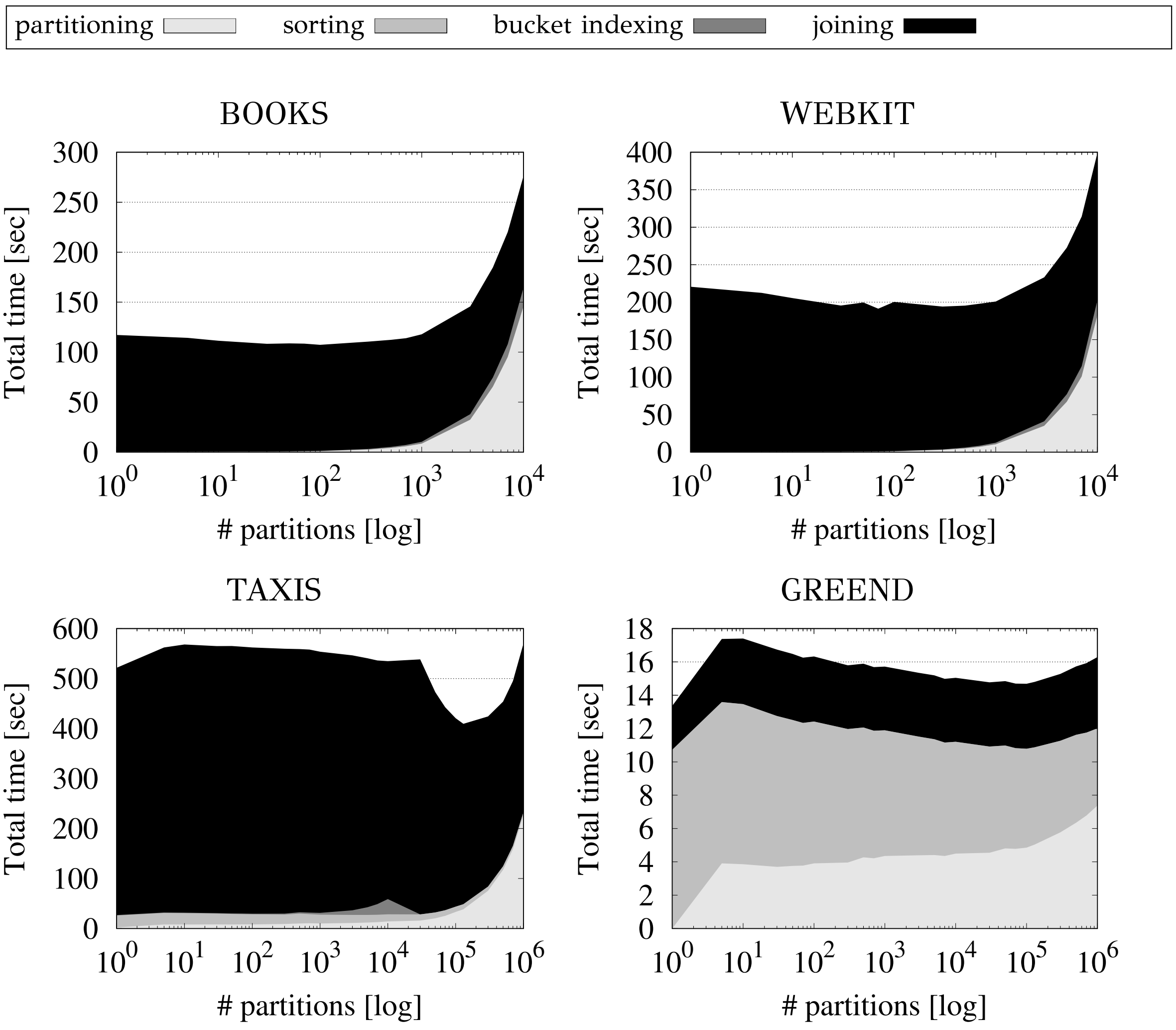


Figure 4.13: Join processing breakdown: unindexed inputs

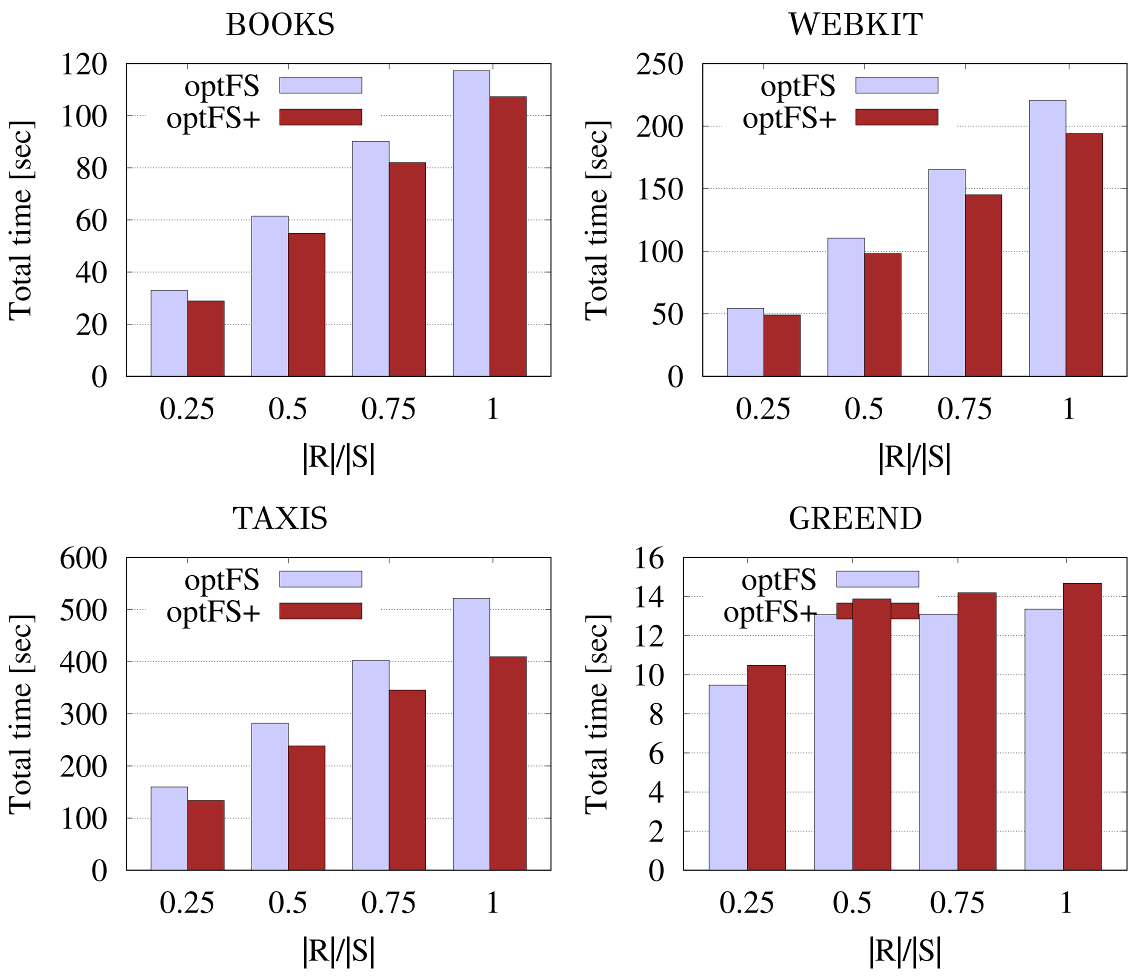


Figure 4.14: Join processing: unindexed inputs

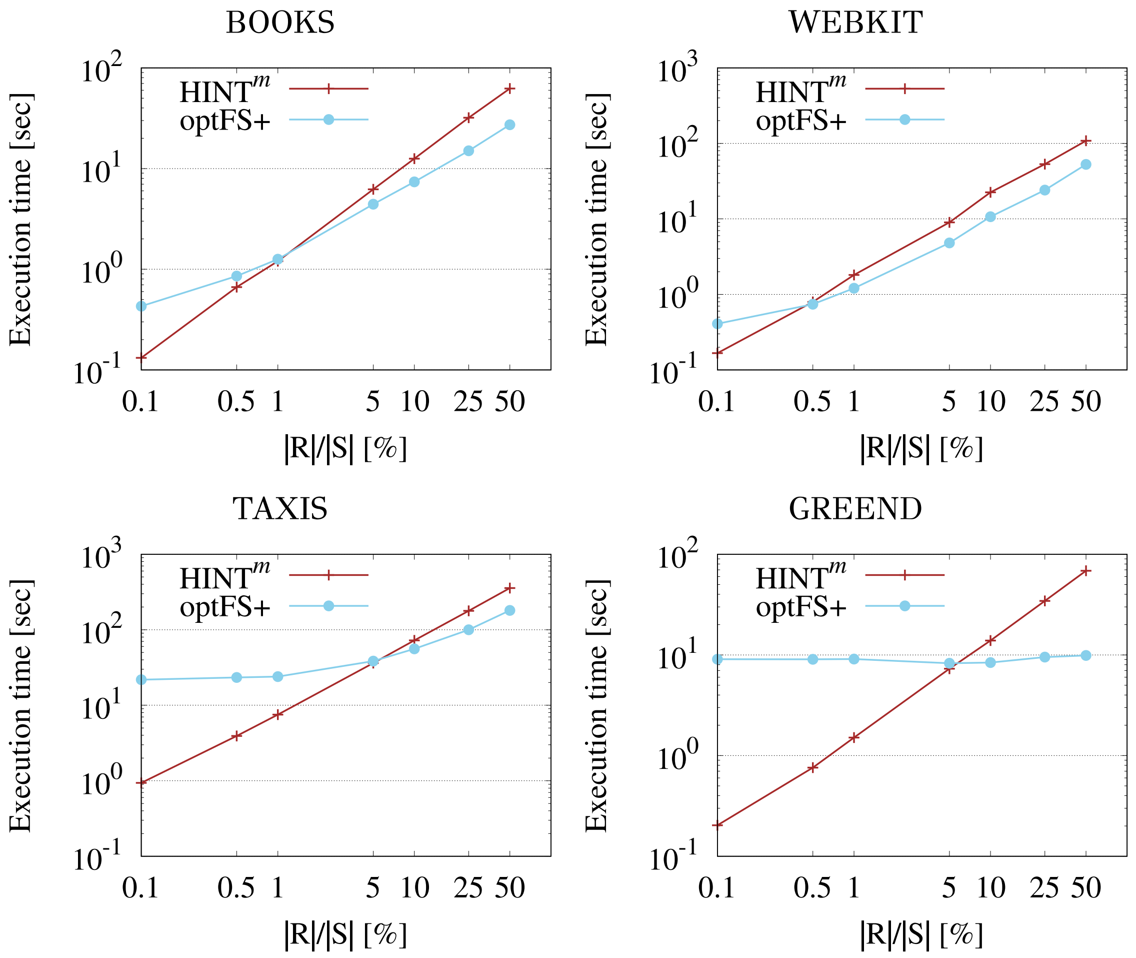


Figure 4.15: G-OVERLAPS based interval joins, real datasets

Chapter 5

Indexing for Allen's algebra

In this chapter, we generalize HINT to efficiently answer queries that involve Allen's predicates, allowing for more complex interval data analysis. The main challenge is to optimize the index for all the query predicates, which access differently the relevant partitions. Another challenge emerges from the increased information we need to keep for each interval, because different endpoints are useful for answering different query predicates. Our index is evaluated against the state-of-the-art solutions with multiple real datasets. Our experiments show a small increase in storage consumption but also a consistent lead in query throughput.

Outline The rest of the chapter is organized as follows. Section 5.1 discusses necessary changes to HINT^m for efficiently evaluating selection queries under the Allen's algebra relationships, and Section 5.2 follows up with the experimental evaluation. Finally, Section 5.3 concludes the chapter.

Table 5.1: Supporting Allen's algebra, setup optimized for `G-OVERLAPS` relationship (Table 4.2)

q **`REL`** s	**definition**	**result set**

`EQUALS`	$q.st = s.st \wedge$
	 $q.end = s.end$
		if $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end = s.end\}$
		 else $\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\} \bigcap \{s \in P^{R_{in}}_{\ell',l} : q.end = s.end\}$

`STARTS`	$q.st = s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end < s.end\} \bigcup \{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\}$ **else** $\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\} \bigcap \{\bigcup_{\forall\ell'} \{\{s \in P^{R_{in}}_{\ell',l} : q.end < s.end\} \bigcup P^{R_{aft}}_{\ell',l}\}\}$
`STARTED_BY`	$q.st = s.st \wedge$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end > s.end\}$ **else** $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st\} \bigcup$ $\{\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\} \bigcap \{\bigcup_{\forall\ell'} \{\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\} \bigcup \{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\}\}\}\}$
`FINISHES`	$q.end = s.end \wedge$ $q.st > s.st$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,l} : q.end = s.end \wedge q.st > s.st\} \bigcup \{s \in P^{R_{in}}_{\ell,l} : q.end = s.end\}$ **else** $\{s \in P^{R_{in}}_{\ell,l} : q.end = s.end\} \bigcap \{\bigcup_{\forall\ell'} \{\{s \in P^{O_{aft}}_{\ell',f} : q.st > s.st\} \bigcup P^{R_{aft}}_{\ell',f}\}\}$
`FINISHED_BY`	$q.end = s.end \wedge$ $q.st < s.st$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,l} : q.end = s.end \wedge q.st < s.st\}$ **else** $\{s \in P^{O_{in}}_{\ell,l} : q.end = s.end\} \bigcup$ $\{\{s \in P^{R_{in}}_{\ell,l} : q.end = s.end\} \bigcap \{\bigcup_{\forall\ell'} \{\{s \in P^{O_{aft}}_{\ell',f} : q.st < s.st\} \bigcup \{\bigcup_{\forall f<i<l} P^{O_{aft}}_{\ell',i}\}\}\}\}$
`MEETS`	$q.end = s.st$	$\forall\ell$: $\{s \in P^{O_{in}}_{\ell,l} \bigcup P^{O_{aft}}_{\ell,l} : q.end = s.st\}$
`MET_BY`	$q.st = s.end$	$\forall\ell$: $\{s \in P^{O_{in}}_{\ell,f} \bigcup P^{R_{in}}_{\ell,f} : q.st = s.end\}$
`OVERLAPS`	$q.st < s.st \wedge$ $q.end > s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st < s.st \wedge q.end > s.st \wedge q.end < s.end\} \bigcup$ $\{s \in P^{O_{aft}}_{\ell,f} : q.st < s.st \wedge q.end > s.st\}$ **else** $\{s \in P^{O_{in}}_{\ell,l} : q.end > s.st \wedge q.end < s.end\} \bigcup \{s \in P^{O_{aft}}_{\ell,l} : q.end > s.st\} \bigcup$ $\{\{\{s \in P^{R_{in}}_{\ell,l} : q.end < s.end\} \bigcup P^{R_{aft}}_{\ell,l}\} \bigcap \{\bigcup_{\forall\ell'} \{\{s \in P^{O_{aft}}_{\ell',f} : q.st < s.st\} \bigcup \{\bigcup_{\forall f<i<l} P^{O_{aft}}_{\ell',i}\}\}\}\}$
`OVERLAPPED_BY`	$q.st > s.st \wedge$ $q.st < s.end \wedge$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end\} \bigcup$ $\{s \in P^{R_{in}}_{\ell,f} : q.st < s.end \wedge q.end > s.end\}$ **else** $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.st < s.end\} \bigcup \{s \in P^{R_{in}}_{\ell,f} : q.st < s.end\} \bigcup$ $\{\{\{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\} \bigcup P^{R_{aft}}_{\ell,f}\} \bigcap \{\bigcup_{\forall\ell'} \{\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\} \bigcup \{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\}\}\}\}$
`CONTAINS`	$q.st < s.st \wedge$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st < s.st \wedge q.end > s.end\}$ **else** $\{s \in P^{O_{in}}_{\ell,f} : q.st < s.st\} \bigcup \{\bigcup_{\forall f<i<l} P^{O_{in}}_{\ell,i}\} \bigcup \{s \in P^{O_{in}}_{\ell,l} : q.end > s.end\} \bigcup$ $\{\{s \in P^{O_{aft}}_{\ell,f} : q.st < s.st\} \bigcup \{\bigcup_{\forall f<i<l} P^{O_{aft}}_{\ell,i}\}\} \bigcap \{\bigcup_{\forall\ell'} \{\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\} \bigcup \{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\}\}\}$
`CONTAINED_BY`	$q.st > s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.end < s.end\} \bigcup \{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\} \bigcup$ $\{s \in P^{R_{in}}_{\ell,f} : q.end < s.end\} \bigcup P^{R_{aft}}_{\ell,f}$ **else** $\{\{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\} \bigcup P^{R_{aft}}_{\ell,f}\} \bigcap \{\bigcup_{\forall\ell'} \{\{s \in P^{R_{in}}_{\ell',l} : q.end < s.end\} \bigcup P^{R_{aft}}_{\ell',l}\}\}$
`BEFORE`	$q.end < s.st$	$\forall\ell$: $\{s \in P^{O_{in}}_{\ell,l} \bigcup P^{O_{aft}}_{\ell,l} : q.end < s.st\} \bigcup \{\bigcup_{\forall i>l} \{P^{O_{in}}_{\ell,i} \bigcup P^{O_{aft}}_{\ell,i}\}\}$
`AFTER`	$q.st > s.end$	$\forall\ell$: $\{s \in P^{O_{in}}_{\ell,f} \bigcup P^{R_{in}}_{\ell,f} : q.st > s.end\} \bigcup \{\bigcup_{\forall i<f} \{P^{O_{in}}_{\ell,i} \bigcup P^{R_{in}}_{\ell,i}\}\}$

5.1 Supporting Allen's Algebra

Table 5.1 (first two columns) summarizes the basic relationships of the algebra, each denoted by q `REL` s, where q is the query interval and s, an interval in the input collection $\mathcal{S}$. Note that the `G-OVERLAPS` selection query from the previous sections identifies every interval s non-disjoint to query q, i.e., a combination of all basic algebra's relationships besides `BEFORE` and `AFTER`.

We study selection queries on Allen's relationships under two setups for our hierarchical indexing. We focus on HINT^m, which exhibits similar performance to the comparison-free HINT but significant lower indexing costs, as our experiments showed in Section 4.3.

5.1.1 Setup Optimized for `G-OVERLAPS`

We start off with the HINT^m setup from Chapter 4 (see Table 4.2), optimized for the `G-OVERLAPS` selection. In what follows, we discuss how queries based on Allen's relationships can be evaluated without any structural changes to the index. Table 5.1 summarizes the set of intervals reported for each selection query.

Relationship `EQUALS`. An `EQUALS` selection determines all input intervals identical to query q, i.e., with $q.end = s.end$ and $q.st = s.st$. To answer such a query, we access two specific index partitions; the first relevant $P_{\ell,f}$ at level ℓ and the last relevant $P_{\ell',l}$, at level ℓ'. [1] Intuitively, these two partitions correspond to the first and last partition where HINT^m would store the query interval q, respectively. We then distinguish between two cases. If q overlaps a single partition, i.e., if $f = l$, we need only the intervals that both *start* and *end* inside this partition, i.e., the $P^{O_{in}}_{\ell,f}$ subdivision. So, we report set $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end = s.end\}$. Otherwise, if $f \neq l$, we report results among the intervals that *start* in the first relevant partition (from $P^{O_{aft}}_{\ell,f}$) and *end* in the last (from $P^{R_{in}}_{\ell',l}$), i.e., set $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\right\} \bigcap \left\{s \in P^{R_{in}}_{\ell',l} : q.end = s.end\right\}$. Note that we cannot directly check $q.end = s.end$ as $P^{O_{aft}}_{\ell,f}$ stores only $s.st$ (and $s.id$).

Relationship `STARTS`. According to Allen's algebra, a `STARTS` selection query reports all intervals that *start* where q does, i.e., with $q.st = s.st$, but outlive its *end*, i.e., with $q.end < s.end$. By construction, HINT^m stores such intervals as originals in the first relevant partition. We consider two cases for every index level ℓ. If $f = l$, we report

[1] In the general case, $\ell \neq \ell'$ holds for levels ℓ and ℓ'.

each interval in the $P^{O_{in}}_{\ell,f}$ subdivision that satisfies both query conditions and each interval in $P^{O_{aft}}_{\ell,f}$ that satisfies only $q.st = s.st$; for the latter intervals, their $s.end$ is by construction after $q.end$. So, we report $\left\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\right\}$. In contrast, if $f \neq l$, the results can only come from the intervals that end after the first relevant partition at current level ℓ, i.e., from $P^{O_{aft}}_{\ell,f}$. But, as subdivisions $P^{O_{aft}}_{\ell,f}$ store only $s.st$ according to Table 4.2, we cannot directly check the $q.end < s.end$ condition. Instead, we rely on the replicas inside the last relevant partition at any index level. Intuitively, if an interval $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st\right\}$ is stored as a replica in the last relevant partition l at a level ℓ', which either (1) ends inside l (i.e., $s \in P^{R_{in}}_{\ell',l}$) but after $q.end$ or (2) outlives the partition (i.e., $s \in P^{R_{aft}}_{\ell',l}$) then $q.end < s.end$ holds for s. The above sets are computed as $\bigcup_{\forall \ell'} \left\{\left\{s \in P^{R_{in}}_{\ell',l} : q.end < s.end\right\} \bigcup P^{R_{aft}}_{\ell',l}\right\}$.

Relationship STARTED_BY. As an inverse to STARTS, a STARTED_BY selection determines all intervals that again *start* at $q.st$ but *end* before $q.end$. Therefore, if $f = l$ holds at a level ℓ, we consider only the intervals that *both* start and end inside the partition, reporting set $\left\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st \wedge q.end > s.end\right\}$. Otherwise, results are found among all originals in f. For the $P^{O_{in}}_{\ell,f}$ subdivision, we directly output $\left\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st\right\}$ as their $s.end$ is by construction before $q.end$. For the intervals in $s \in P^{O_{aft}}_{\ell,f}$ with $q.st = s.st$, we apply a similar technique to STARTS for checking the $q.end > s.end$ condition. Intuitively, such an interval s will be reported if it ends at any level ℓ', either inside a partition i with $f < i < l$ or in the last relevant partition l but before $q.end$. For this purpose, we check if s is inside set $\bigcup_{\forall \ell'} \left\{\left\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\right\} \bigcup \left\{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\right\}\right\}$.

Relationship FINISHES. This selection query returns all intervals that *end* exactly where query q does, i.e., with $q.end = s.end$, but *start* before q, i.e., with $q.st > s.st$. If q overlaps a single partition ($f = l$) at a level ℓ, we consider the intervals that end in the last relevant partition l: $\left\{s \in P^{O_{in}}_{\ell,l} : q.end = s.end \wedge q.st > s.st\right\} \bigcup \left\{s \in P^{R_{in}}_{\ell,l} : q.end = s.end\right\}$. Otherwise ($f \neq l$), only replicas that end inside partition l (Subdivision $P^{R_{in}}_{\ell,l}$) with $q.end = s.end$ can be part of the results. To this end, we face a similar challenge to STARTS/STARTED_BY as $P^{R_{in}}_{\ell,l}$ does not store $s.st$ (see Table 4.2) to directly check $q.st > s.st$. The solution is to check if an interval $\left\{s \in P^{R_{in}}_{\ell,l} : q.end = s.end\right\}$ is contained in set $\bigcup_{\forall \ell'} \left\{\left\{s \in P^{O_{aft}}_{\ell',f} : q.st > s.st\right\} \bigcup P^{R_{aft}}_{\ell',f}\right\}$, i.e., the intervals that either (1) start before $q.st$ in the first relevant partition f at any level ℓ' or (2) are stored in $P^{R_{aft}}_{\ell',f}$ and so, their start is by construction before $q.st$.

Relationship `FINISHED_BY`. A `FINISHED_BY` selection inverses the second condition of `FINISHES`, determining intervals with $q.end = s.end$ and $q.st < s.st$. For a level ℓ, if $f = l$ we report the intervals that start and end inside the partition, and satisfy both conditions, i.e., set $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st\right\}$. Otherwise $(f \neq l)$, the results are among all intervals that end in partition l, i.e., set $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end\right\}$. For the intervals from subdivision $P_{\ell,l}^{O_{in}}$, $q.st < s.st$ holds by construction while for $P_{\ell,l}^{R_{in}}$ intervals, a direct check of the condition is not possible. Instead, we check such an interval s against the set of intervals that start either (1) after q in the first relevant partition at any level ℓ' or (2) inside the partitions in between the first and the last relevant; set $\bigcup_{\forall \ell'} \left\{\left\{s \in P_{\ell',f}^{O_{aft}} : q.st < s.st\right\} \bigcup \left\{\bigcup_{\forall f<i<l} P_{\ell',i}^{O_{aft}}\right\}\right\}$.

Relationship `MEETS`. This selection query returns all intervals that *start* at $q.end$. Under this, we report for each level ℓ, all originals in the last relevant partition l that satisfy the $q.end = s.st$ condition, i.e., set $\left\{s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end = s.st\right\}$.

Relationship `MET_BY`. This selection query returns all intervals that *end* at $q.st$. To this end, the results are among the intervals that end inside the first relevant partition f, i.e., set $\left\{s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st = s.end\right\}$, at each level ℓ.

Relationship `OVERLAPS`. An `OVERLAPS` selection determines all non-disjoint intervals to query q, which start *after* $q.st$ and end *after* $q.end$. If q overlaps a single partition $(f = l)$ at a level ℓ, such intervals are found among the originals in the partition; for the $P_{\ell,f}^{O_{in}}$ subdivision all query conditions are checked, while for an s in $P_{\ell,f}^{O_{aft}}$, $q.end < s.end$ always holds. So, we report set $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.st \wedge q.end < s.end\right\}$ $\bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.st\right\}$. Otherwise, results are reported in two parts. The first part is drawn from the originals in the last relevant partition at each level ℓ, i.e., $\left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{O_{aft}} : q.end > s.st\right\}$. For the second part, we consider the intervals that start before partition l and outlive q, i.e., set $\left\{\left\{s \in P_{\ell,l}^{R_{in}} : q.end < s.end\right\} \bigcup P_{\ell,l}^{R_{aft}}\right\}$. For every such interval s, $q.end > s.st$ holds by construction, but we need to check its start against $q.st$. As subdivisions $P_{\ell}^{R_{in}}$ and $P_{\ell}^{R_{aft}}$ do not store $s.st$, we cannot directly check the $q.st < s.st$ condition. Instead, we compare s against all $P_{\ell'}^{O_{aft}}$ at any level ℓ' that (1) either start before $q.st$ in the first relevant partition f or (2) inside every partition in between f and l, i.e., set $\bigcup_{\forall \ell'} \left\{\left\{s \in P_{\ell',f}^{O_{aft}} : q.st < s.st\right\} \bigcup \left\{\bigcup_{\forall f<i<l} P_{\ell',i}^{O_{aft}}\right\}\right\}$.

Relationship `OVERLAPPED_BY`. As inverse to `OVERLAPS`, the `OVERLAPPED_BY` selection determines all non-disjoint intervals to q that start *before* $q.st$ and end *before* $q.end$.

If $f = l$, we draw the results from all intervals (both originals and replicas) that *end* inside the partition; set $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end\}$ $\bigcup \{s \in P^{R_{in}}_{\ell,f} : q.st < s.end \wedge q.end > s.end\}$. Otherwise, the results consist of two parts for every level ℓ. The first part includes again originals and replicas that end inside the first relevant partition f, but now, condition $q.end > s.end$ always holds by construction. Hence, we report set $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.st < s.end\} \bigcup$ $\{s \in P^{R_{in}}_{\ell,f} : q.st < s.end\}$. For the second part, we seek results among all intervals that start before q, i.e., originals $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\right\}$ and replicas $P^{R_{aft}}_{\ell,f}$ for both sets $q.st < s.end$ holds by construction as intervals outlive the first relevant partition f. As neither of the $P^{O_{aft}}_{\ell,f}$ and $P^{R_{aft}}_{\ell,f}$ subdivisions maintains $s.end$, we check $q.end > s.end$ by determining the replicas at any index level ℓ' that end (1) either before the last relevant partition l or (2) inside l after $q.end$, i.e., set $\bigcup_{\forall \ell'} \left\{\left\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\right\} \bigcup \{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\}\right\}$.

Relationship CONTAINS. This selection query returns all intervals, fully *contained* inside the query interval q, i.e., with $q.st < s.st \wedge q.end > s.end$. For every level ℓ, if $f = l$, q can contain only intervals that *both* start and end in this partition, i.e., from subdivision $P^{O_{in}}_{\ell,f}$; we report set $\{s \in P^{O_{in}}_{\ell,f} : q.st < s.st \wedge q.end > s.end\}$. Otherwise, the results are drawn from the original intervals in every partition from the first relevant partition f to the last l; for the latter only originals that end inside the partition are considered. Specifically, for the intervals in $P^{O_{in}}_{\ell}$ subdivisions, we report $\{s \in P^{O_{in}}_{\ell,f} : q.st < s.st\} \bigcup \left\{\bigcup_{\forall f<i<l} P^{O_{in}}_{\ell,i}\right\} \bigcup \{s \in P^{O_{in}}_{\ell,l} : q.end > s.end\}$; observe how only one condition is checked for partitions f and l, while for every partition i in between, all originals that end inside i are directly output. In contrast, for all intervals in the $P^{O_{aft}}_{\ell}$ subdivisions, we need to check the $q.end > s.end$ condition; additionally, for every $s \in P^{O_{aft}}_{\ell,f}$ subdivision, we also check if $q.st < s.st$ holds. As $P^{O_{aft}}_{\ell}$ subdivisions store only $s.st$, $q.end < s.end$ is checked similarly to OVERLAPPED_BY, i.e., using set $\bigcup_{\forall \ell'} \left\{\left\{\bigcup_{\forall f<i<l} P^{R_{in}}_{\ell',i}\right\} \bigcup \{s \in P^{R_{in}}_{\ell',l} : q.end > s.end\}\right\}$.

Relationship CONTAINED_BY. This selection determines all intervals that fully *contain* q, i.e., with $q.st > s.st \wedge q.end < s.end$. For each level ℓ, if $f = l$, the result intervals are found among all subdivisions in the partition, reporting $\{s \in P^{O_{in}}_{\ell,f} : q.st > s.st \wedge q.end < s.end\} \bigcup \left\{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\right\} \bigcup$ $\{s \in P^{R_{in}}_{\ell,f} : q.end < s.end\} \bigcup P^{R_{aft}}_{\ell,f}$. In contrast, if $f \neq l$, the results are among the intervals that (1) start before $q.st$, corresponding to set $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st > s.st\right\} \bigcup P^{R_{aft}}_{\ell,f}$, and (2) end after $q.end$. As the $P_{\ell}^{O_{aft}}$ or the $P_{\ell}^{R_{aft}}$ subdivisions do not store $s.end$, in order to check the $q.end < s.end$ condition, we need to intersect the above candidates set with the replicas at any level ℓ' that either end inside the last relevant partition l or outlive it, i.e., set $\bigcup_{\forall \ell'} \left\{ \{ s \in P_{\ell',l}^{R_{in}} : q.end < s.end \} \bigcup P_{\ell',l}^{R_{aft}} \right\}$.

Relationship `BEFORE`. A `BEFORE` selection determines all intervals that start *after* q. Such intervals are found at each level ℓ as originals either (1) inside the last relevant partition l, if they satisfy $q.end < s.st$, i.e., set $\left\{ s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end < s.st \right\}$ or (2) inside every partition after l, i.e., set $\bigcup_{\forall i > l} \left\{ P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{O_{aft}} \right\}$. Note that replicas from these partitions are ignored as they will only produce duplicate results.

Relationship `AFTER`. An `AFTER` selection determines all intervals that end *before* q. Results are found at each level among the intervals which end inside either (1) the first relevant partition f and satisfy $q.st > s.end$, i.e., set $\{ s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st > s.end \}$ or (2) every partition before f, i.e., set $\bigcup_{\forall i < f} \{ P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{R_{in}} \}$. Note that subdivisions $P_{\ell,i}^{O_{aft}}$ and $P_{\ell,i}^{R_{aft}}$ are ignored to avoid duplicate results.

Table 5.2: Allen's algebra relationships, 'One setup for all'

q REL s	**definition**	**result set**

EQUALS	$q.st = s.st \wedge$
	$q.end = s.end$
		if $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end = s.end\right\}$
		else $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end = s.end\right\}$

STARTS	$q.st = s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end < s.end\right\}$
STARTED_BY	$q.st = s.st \wedge$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st = s.st \wedge q.end > s.end\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{in}} : q.st = s.st\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st = s.st \wedge q.end > s.end\right\}$
FINISHES	$q.end = s.end \wedge$ $q.st > s.st$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st > s.st\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end\right\}$ **else** $\left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st > s.st\right\}$
FINISHED_BY	$q.end = s.end \wedge$ $q.st < s.st$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end \wedge q.st < s.st\right\}$ **else** $\left\{s \in P_{\ell,l}^{O_{in}} : q.end = s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{in}} : q.end = s.end \wedge q.st < s.st\right\}$
MEETS	$q.end = s.st$	$\forall\ell$: $\left\{s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end = s.st\right\}$
MET_BY	$q.st = s.end$	$\forall\ell$: $\left\{s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st = s.end\right\}$
OVERLAPS	$q.st < s.st \wedge$ $q.end > s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.st \wedge q.end < s.end\right\} \bigcup$ $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.st\right\}$ **else** $\left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{O_{aft}} : q.end > s.st\right\} \bigcup$ $\left\{s \in P_{\ell,l}^{R_{in}} : q.st < s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,l}^{R_{aft}} : q.st < s.st\right\}$
OVERLAPPED_BY	$q.st > s.st$ $q.st < s.end$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end \wedge q.end > s.end\right\} \bigcup$ $\left\{s \in P_{\ell,f}^{R_{in}} : q.st < s.end \wedge q.end > s.end\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.st < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{in}} : q.st < s.end\right\} \bigcup$ $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end > s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{aft}} : q.end > s.end\right\}$
CONTAINS	$q.st < s.st \wedge$ $q.end > s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st \wedge q.end > s.end\right\}$ **else** $\left\{s \in P_{\ell,f}^{O_{in}} : q.st < s.st\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \wedge q.end > s.end\right\} \bigcup$ $\left\{\bigcup_{\forall f<i<l} P_{\ell,i}^{O_{in}}\right\} \bigcup \left\{s \in \bigcup_{\forall f<i<l} P_{\ell,i}^{O_{aft}} : q.end > s.end\right\} \bigcup$ $\left\{s \in P_{\ell,l}^{O_{in}} : q.end > s.end\right\}$
CONTAINED_BY	$q.st > s.st \wedge$ $q.end < s.end$	$\forall\ell$: **if** $f = l$, $\left\{s \in P_{\ell,f}^{O_{in}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st\right\} \bigcup$ $\left\{s \in P_{\ell,f}^{R_{in}} : q.end < s.end\right\} \bigcup P_{\ell,f}^{R_{aft}}$ **else** $\left\{s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \wedge q.end < s.end\right\} \bigcup \left\{s \in P_{\ell,f}^{R_{aft}} : q.end < s.end\right\}$
BEFORE	$q.end < s.st$	$\forall\ell$: $\left\{s \in P_{\ell,l}^{O_{in}} \bigcup P_{\ell,l}^{O_{aft}} : q.end < s.st\right\} \bigcup \left\{\bigcup_{\forall i>l} \left\{P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{O_{aft}}\right\}\right\}$
AFTER	$q.st > s.end$	$\forall\ell$: $\left\{s \in P_{\ell,f}^{O_{in}} \bigcup P_{\ell,f}^{R_{in}} : q.st > s.end\right\} \bigcup \left\{\bigcup_{\forall i<f} \left\{P_{\ell,i}^{O_{in}} \bigcup P_{\ell,i}^{R_{in}}\right\}\right\}$

5.1.2 One Setup for All

The storage optimization discussed in Section 4.2.1 allows the G-OVERLAPS setup of HINT^m to reduce the memory footprint of the index and improve cache locality. But as an optimization technique tailored for the G-OVERLAPS relationship, it has a negative impact on Allen's algebra basic relationships. The key issue is that we cannot directly check the conditions on $s.end$ for the $P^{O_{aft}}$ and $P^{R_{aft}}$ subdivisions and on $s.st$ for $P^{R_{in}}$ and $P^{R_{aft}}$. Instead, we are forced to access extra partitions to implicitly conduct these checks, e.g., the $P^{R_{in}}_{\ell',l}$ and $P^{R_{aft}}_{\ell',l}$ subdivisions in the last relevant partition l at each index level ℓ', for the STARTS relationship.

In view of this shortcoming, we next consider a *subs+sort* setup of HINT^m for Allen's algebra.[2] Essentially, no changes are required if query q overlaps a single partition ($f = l$) at a level ℓ as all necessary information is available for the selection conditions. Further, the computation of MEETS, MET_BY, BEFORE and AFTER queries remains unchanged. Hence, in what follows, we discuss the necessary changes for the rest of relationships in the $f \neq l$ case.

Relationship EQUALS. We can now directly retrieve results from the first relevant partition f and the $P^{O_{aft}}_{\ell,f}$ subdivision by checking both query conditions, i.e., we report set $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st \wedge q.end = s.end\right\}$.

Relationship STARTS. With $s.end$ in $P^{O_{aft}}_{\ell,f}$, both query conditions can be directly checked at each level ℓ and thus report $\left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st \wedge q.end < s.end\right\}$.

Relationship STARTED_BY. Similar to STARTS, we can directly check both conditions for $P^{O_{aft}}_{\ell,f}$ in the first relevant partition f. We report $\{s \in P^{O_{in}}_{\ell,f} : q.st = s.st\} \bigcup \left\{s \in P^{O_{aft}}_{\ell,f} : q.st = s.st \wedge q.end > s.end\right\}$, at each level.

Relationship FINISHES. With $s.st$ in $P^{R_{in}}_{\ell,l}$ subdivisions, we can directly check $q.st > s.st$ and report $\{s \in P^{R_{in}}_{\ell,l} : q.end = s.end \wedge q.st > s.st\}$, at each level.

Relationship FINISHED_BY. Similar to FINISHES, we can directly check both conditions on $P^{R_{in}}_{\ell,l}$ and thus, report at each level ℓ, set $\{s \in P^{O_{in}}_{\ell,l} : q.end = s.end\} \bigcup \{s \in P^{R_{in}}_{\ell,l} : q.end = s.end \wedge q.st < s.st\}$.

Relationship OVERLAPS. With $s.st$ in subdivisions $P^{R_{in}}_{\ell}$ and $P^{R_{aft}}_{\ell}$, we directly check $q.st < s.st$ for partition l. So, we report $\{s \in P^{R_{in}}_{\ell,l} : q.st < s.st \wedge q.end < s.end\}$

[2]The cache misses and the skewness & sparsity optimizations are orthogonal and can be straightforwardly combined with the $_{\text{subs+sort}}$HINT^m setup.

$\bigcup \left\{ s \in P_{\ell,l}^{R_{aft}} : q.st < s.st \right\}$ intervals at each level along with the set $\left\{ s \in P_{\ell,l}^{O_{in}} : q.end > s.st \land q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,l}^{O_{aft}} : q.end > s.st \right\}$.

Relationship **`OVERLAPPED_BY`.** With $s.end$ stored in $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$, we can directly check $q.end > s.end$, reporting set $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \land q.end > s.end \right\}$ $\bigcup \left\{ s \in P_{\ell,f}^{R_{aft}} : q.end > s.end \right\}$ along with the intervals contained in $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st > s.st \land q.st < s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{R_{in}} : q.st < s.end \right\}$.

Relationship **`CONTAINS`.** With $s.end$ in $P_{\ell}^{O_{aft}}$ subdivisions, we can directly check the $q.end > s.end$ condition to output $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st < s.st \land q.end > s.end \right\} \bigcup \left\{ s \in \bigcup_{\forall f<i<l} P_{\ell,i}^{O_{aft}} : q.end > s.end \right\}$ along with the set $\left\{ s \in P_{\ell,f}^{O_{in}} : q.st < s.st \right\} \bigcup \left\{ s \in P_{\ell,l}^{O_{in}} : q.end > s.end \right\}$ $\bigcup \left\{ \bigcup_{\forall f<i<l} P_{\ell,i}^{O_{in}} \right\}$ from $P_{\ell}^{O_{in}}$ subdivisions at each level.

Relationship **`CONTAINED_BY`.** With $s.end$ stored in both $P_{\ell,f}^{O_{aft}}$ and $P_{\ell,f}^{R_{aft}}$ subdivisions, we can now directly check the $q.end < s.end$ condition at each level ℓ, reporting the intervals $\left\{ s \in P_{\ell,f}^{O_{aft}} : q.st > s.st \land q.end < s.end \right\} \bigcup \left\{ s \in P_{\ell,f}^{R_{aft}} : q.end < s.end \right\}$.

5.1.3 Bottom-up Evaluation Approach

Both setups of HINT^m can benefit from the bottom-up approach in Section 4.1.2. The idea is to determine the levels when the last bit of the first (last) relevant partition f (l) are set to 1 or 0, for the first time. Due to lack of space, we discuss only `STARTS` for the `G-OVERLAPS` setup as an example. Specifically, results are found among the original intervals stored in the first relevant partition f up to the level where the last bit in f is 1, for the first time. All originals in f at a higher level start by construction of the index before $q.st$ and thus, violate $q.st = s.st$. In addition, at levels after the one where the last bit of l is 0 for the first time, $q.end < s.end$ always holds for all $s \in P_{\ell',l}^{R_{in}}$. Consider for example the query q in Figure 4.3. Candidate results are contained only as originals in $P_{4,5}$, where the last bit of $f = 5$ is 1. Also as the last bit of l is 0 at the 4th level, all $P^{R_{in}}$ intervals in $P_{2,2}$, $P_{1,1}$, $P_{0,0}$ satisfy $q.end < s.end$.

5.2 Experiments on Allen's Algebra

For the second part of our experiments, we focus on selection queries under the basic relationships of Allen's algebra. We first compare the two alternative HINT^m setups from Section 5.1 and then put the best setup against the competition. We extended our code for all competitive indices in Section 4.3 to support Allen's algebra. We ran our tests on datasets BOOKS, WEBKIT, TAXIS and GREEND. Lastly, parameter m and all other index parameters are set according to Table 4.6.

5.2.1 Determining the Best Index Setup

Figure 5.1 reports the throughputs achieved by the two HINT^m setups; results in WEBKIT and GREEND are similar and therefore omitted due to lack of space. Note that both setups adopt the bottom-up evaluation (Section 4.1.2) and employ the *skewness & sparsity* and the *cache misses* optimizations (Sections 4.2.2 and 4.2.3). The results back up our discussion in Section 5.1. The 'one setup for all' setup drastically improves the performance of HINT^m for the majority of the queries. Essentially, the `G-OVERLAPS` setup matches the performance of 'one setup for all' in the `G-OVERLAPS` relationship, as expected, and in relationships where only one partition per level is examined by both setups, without the need to indirectly check a condition, i.e., in `MEETS`, `MET_BY`, `BEFORE` and `AFTER`. In the rest, 'one setup for all' is from one to several orders of magnitude faster. For the rest of our analysis, HINT^m always operates under 'one setup for all'.

5.2.2 Index Performance Comparison

Figure 5.2 compares the performance of all studied indices. The first 4 rows of plots report the results for `OVERLAPS`, `OVERLAPPED_BY`, `CONTAINS`, `CONTAINED_BY`, while varying the query extent, similar to Figure 4.10. Note that for `CONTAINED_BY` on TAXIS and GREEND, we consider a different range of values because these datasets contain significantly shorter intervals compared to BOOKS and WEBKIT. The last row of plots reports the throughput on the rest of the relationships where the selection queries essentially resemble typical stabbing queries, i.e., query overlaps either one partition per level or only two overall specific partitions in `EQUALS`.

Overall, HINT^m exhibits the highest throughput for all queries based on Allen's algebra relationships, in line with the results in Figure 4.10. Its performance gap to the competitor indices ranges from almost half to several orders of magnitude. Essentially, the smallest performance gap are observed mainly in WEBKIT and GREEND where the input intervals are very short.

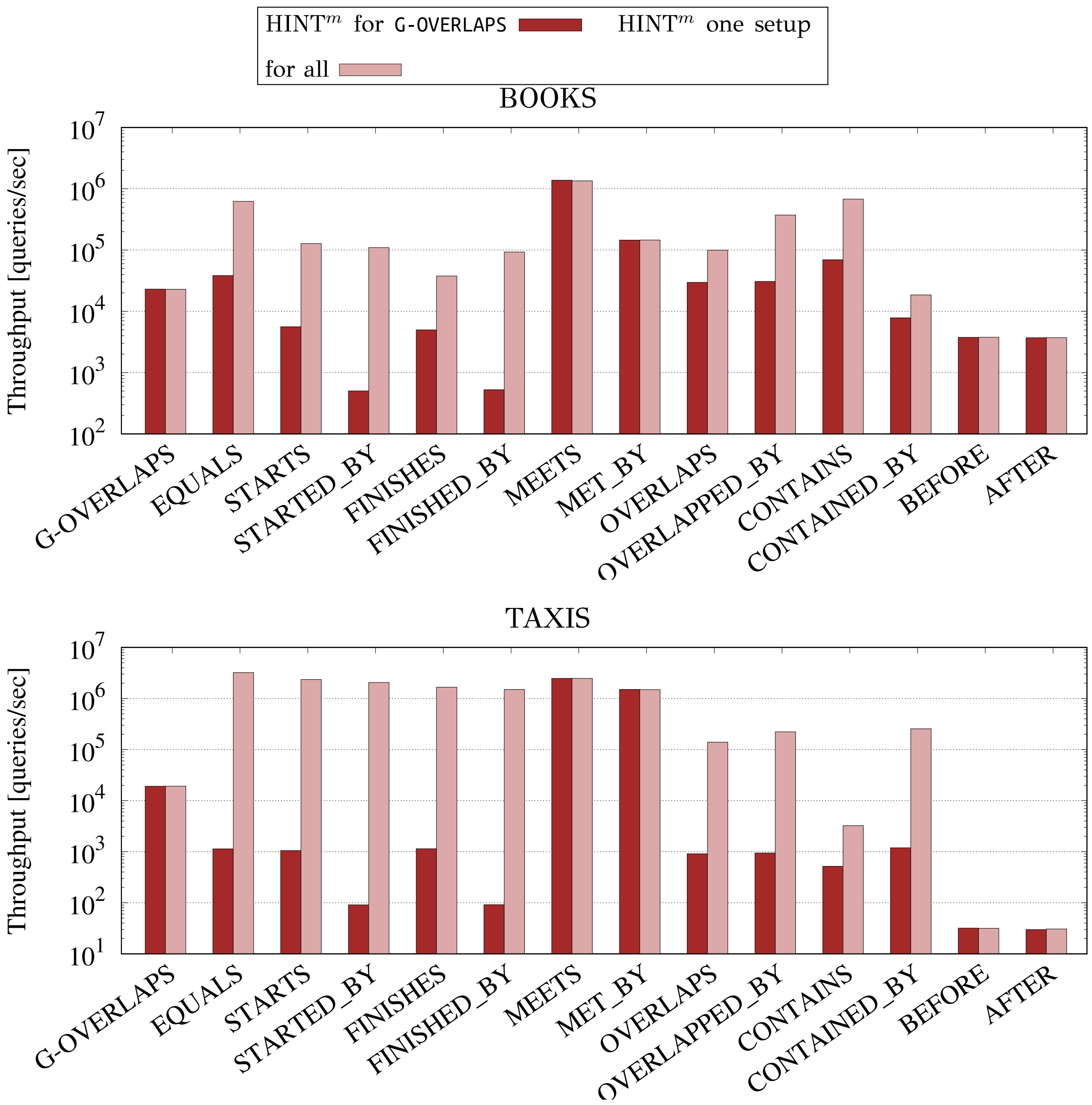
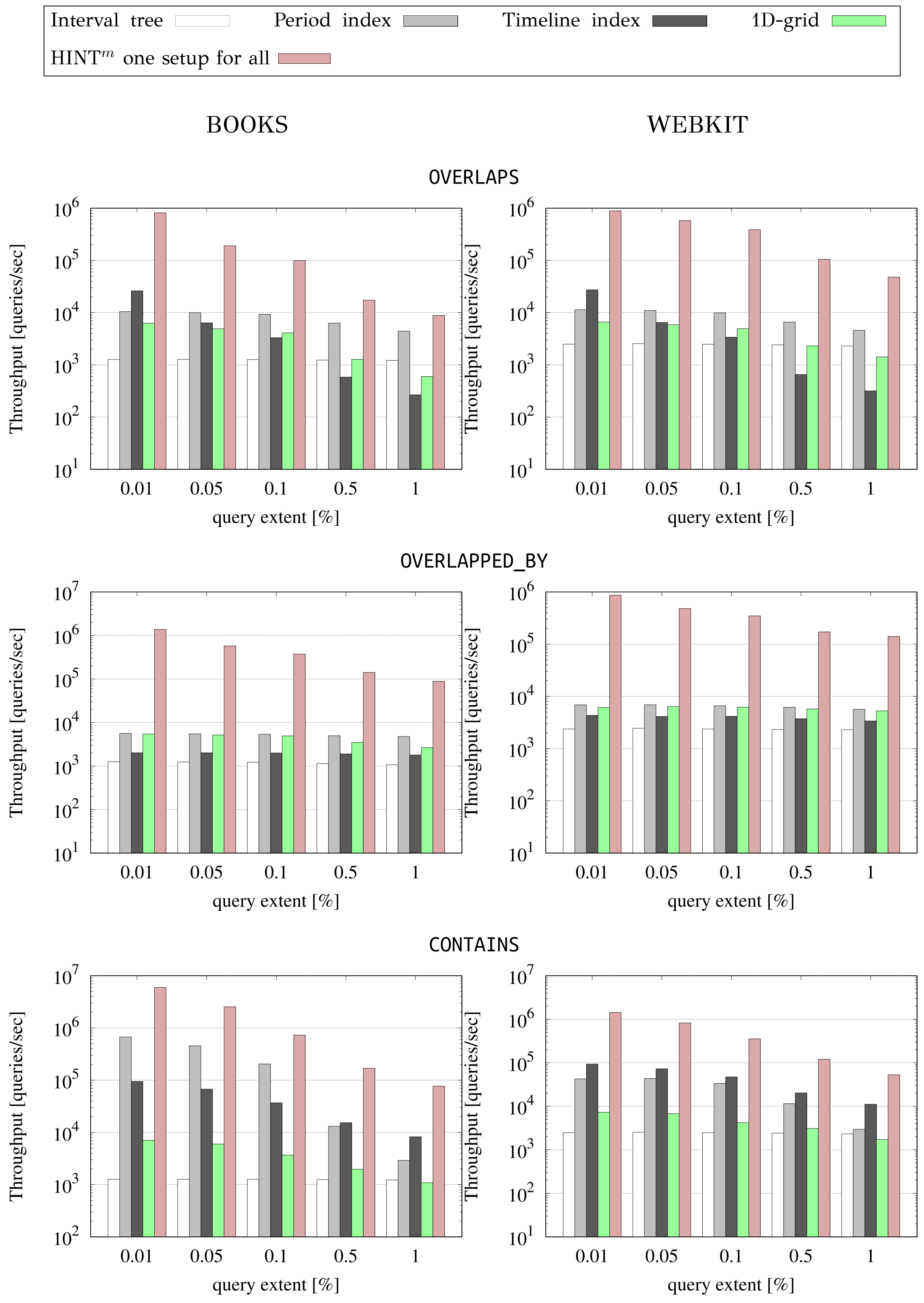
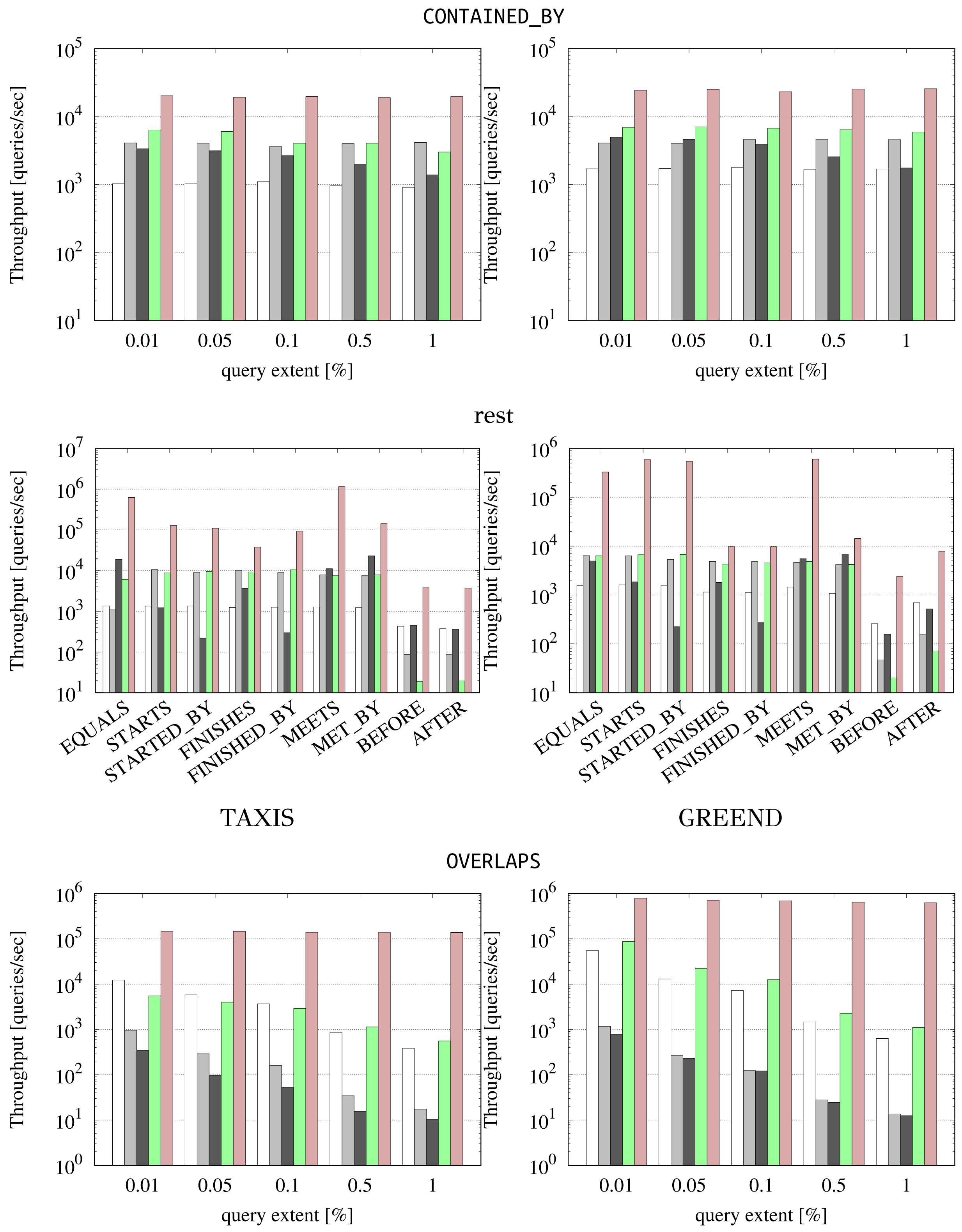


Figure 5.1: Comparing HINT^m setups

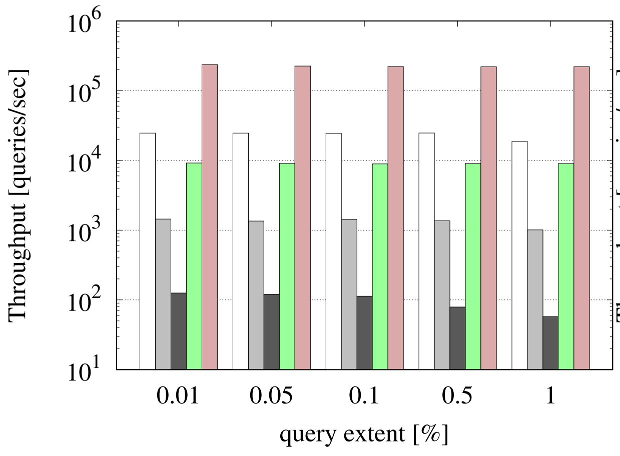
5.3 Conclusions

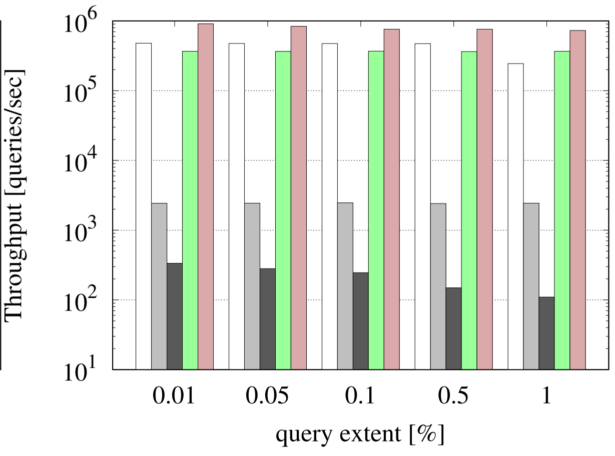
We generalized HINT, which fully supports selection queries based on Allen's relationships [72] between intervals, achieving consistently excellent performance independently of the query predicate. Our experimental analysis on real and synthetic datasets shows that HINT outperforms previous work by one order of magnitude in a wide variety of interval data and query distributions.


Interval tree
Period index
Timeline index
1D-grid
HINT^m one setup for all
BOOKS
WEBKIT
OVERLAPS
OVERLAPPED_BY
CONTAINS
Throughput [queries/sec]
query extent [%]
0.01
0.05
0.1
0.5
1

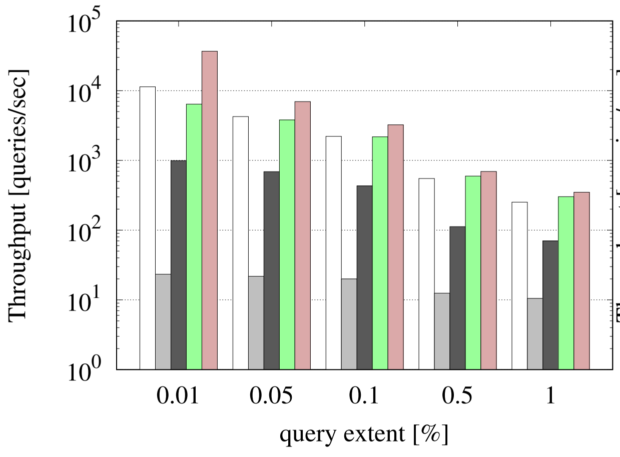

CONTAINED_BY
Throughput [queries/sec]
query extent [%]
0.01
0.05
0.1
0.5
1
rest
EQUALS
STARTS
STARTED_BY
FINISHES
FINISHED_BY
MEETS
MET_BY
BEFORE
AFTER
TAXIS
GREEND
OVERLAPS

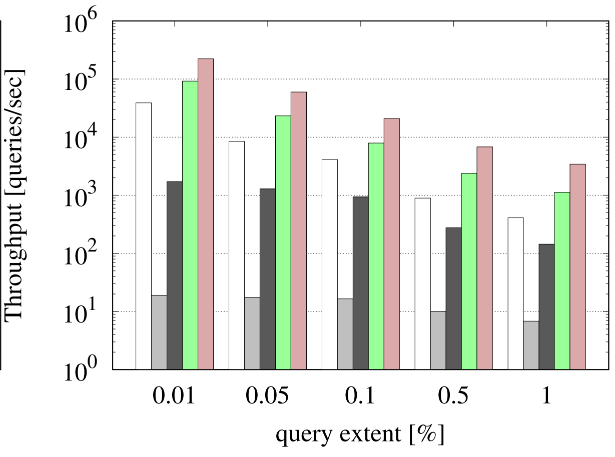
OVERLAPPED_BY



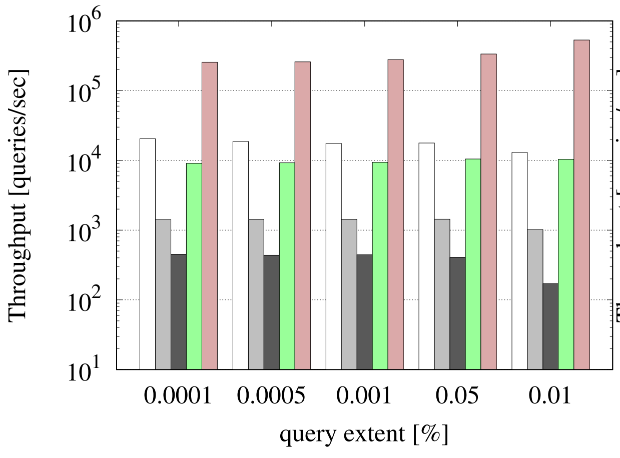


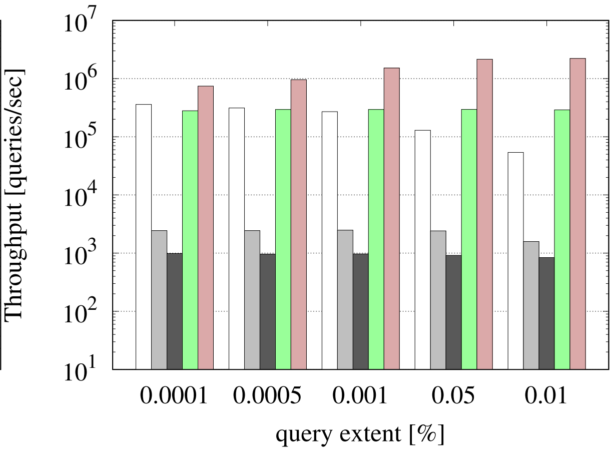
CONTAINS





CONTAINED_BY





rest

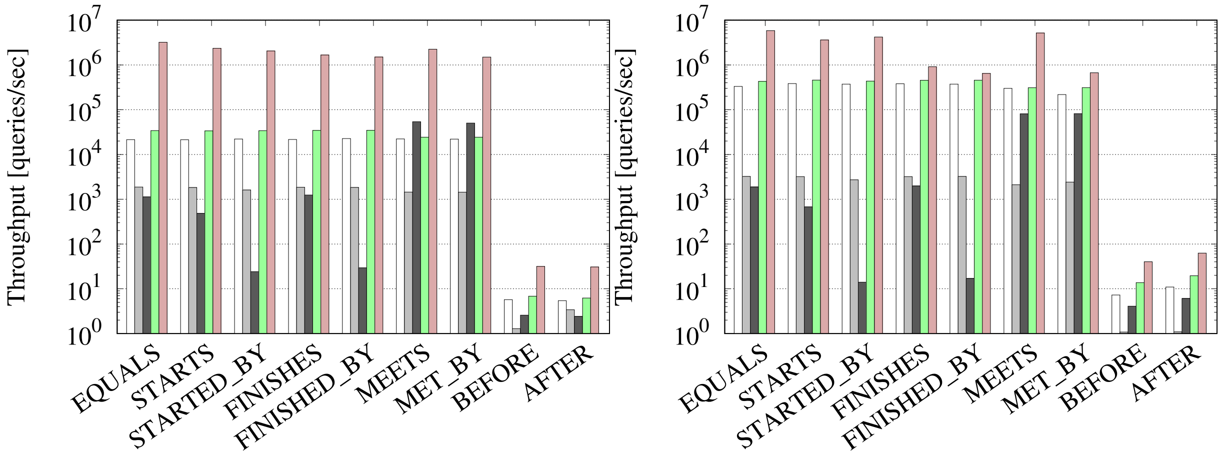


Figure 5.2: Comparing throughputs on Allen's algebra, real datasets

Chapter 6

Indexing Intervals for Transaction Time Temporal Databases

In this chapter, we study the problem of transaction time indexing, i.e., indexing versions of a database table in an evolving database. We propose LIT, a hybrid index, which decouples the management of the current and past states of the indexed column. LIT includes optimized indexing modules for dead and live records, which support efficient queries and updates, and gracefully combines them. We experimentally show that LIT is orders of magnitude faster than the state-of-the-art temporal indices that index live and dead versions simultaneously. In addition, we demonstrate that LIT uses linear space to the number of record versions that it indexes, making it suitable for main-memory temporal data management.

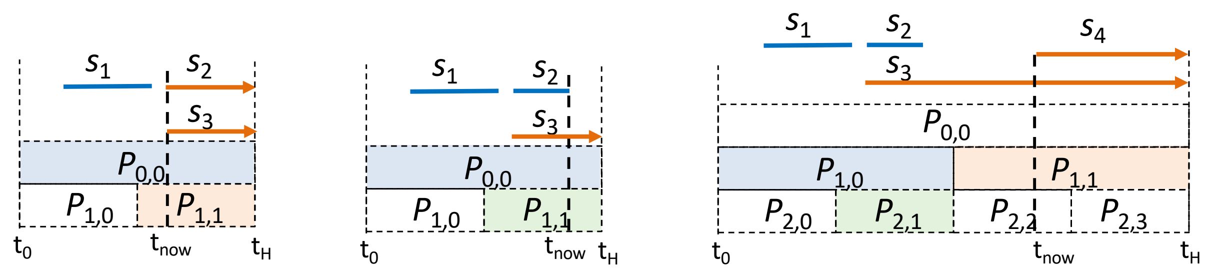
Outline The rest of the chapter is organized as follows. Section 6.1 proposes an extension to HINT, to manage live and dead record versions in an ever-growing time domain. In Section 6.2, we present LIT, the main proposal of this chapter for pure time-travel queries. Section 6.3 discusses how LIT can be extended to index an attribute A of the records besides their temporal validity intervals, in order to support range time-travel queries. Section 6.4 discusses the integration of our main-memory LIT in a DMBS that should support persistence and fault-tolerance (recovery). Section 6.5 includes our experimental analysis and, finally, Section 6.6 concludes the chapter.

6.1 Time-evolving HINT

A first attempt to define an efficient in-memory index for time-evolving tables is to convert HINT [40], the state-of-the-art interval index, to a single data structure that can handle both live and dead intervals (records). We call this data structure *time-evolving* HINT (*te*-HINT). A *te*-HINT for pure time-travel queries extends HINT in two directions. First, it includes both live and dead records, whereas HINT indexes only intervals for which the *end* endpoint is immutable. Second, it supports an evolving domain for the interval endpoints (i.e., an evolving time domain); the original HINT requires a pre-defined domain. These differences require some structural changes and new update operations in *te*-HINT, compared to HINT [40], which are described next.

6.1.1 Live and dead sub-partitions

The first difference between *te*-HINT and HINT is the introduction of *live* partitions in the former. Recall from Chapter 4 that in each partition $P_{\ell,i}$ at level ℓ of HINT, the intervals are divided into two classes: the set of *originals* $P^O_{\ell,i}$ which start inside the domain range of $P_{\ell,i}$ and the set of *replicas* $P^R_{\ell,i}$, which start before the domain of $P_{\ell,i}$. In *te*-HINT, we further classify each interval $s \in P^O_{\ell,i}$ as *live original* or *dead original*, depending on whether its end time point is known; we denote the sub-partitions that hold live and dead originals by $P^{O_L}_{\ell,i}$ and $P^{O_D}_{\ell,i}$, respectively. Similarly, we maintain sub-partitions $P^{R_L}_{\ell,i}$ and $P^{R_D}_{\ell,i}$ for the replicas of $P_{\ell,i}$. Dead intervals in $P^{O_D}_{\ell,i}$ or $P^{R_D}_{\ell,i}$ are immutable, which means that they persist in the partition and cannot move to other partitions, whereas live intervals can be deleted or moved to other partitions.



(a) insert s_2 & s_3 (b) delete s_2 (c) insert s_4

Figure 6.1: Example of *te*-HINT

6.1.2 Handling updates

There are two types of update events over time: either the creation of a new live interval (as a result of an insertion/modification to the database), or the finalization of an existing live interval (as a result of a deletion/modification to the database).

Insertion events. In the case where an insertion event arrives, i.e., a new *live* interval s begins corresponding to a version of a record r, we insert s to *te*-HINT (in live sub-partitions) using the insertion algorithm of HINT [40], assuming that the end time point of s is the end of the current domain of *te*-HINT (i.e., a timepoint in the future), called the *horizon* of *te*-HINT and denoted by t_H. At the same time we insert an entry $\langle r.id, s.start\rangle$ in an auxiliary key-value data structure $\mathcal{H}_{r.id \rightarrow start}$ that facilitates finding a live interval in *te*-HINT given the corresponding record id. Figure 6.1(a) shows a simple example of a 2-level *te*-HINT, holding interval s_1, which corresponds to a dead record, in partition $P_{0,0}$ (sub-partition $P_{0,0}^{O_D}$). Two new live intervals s_2 and s_3 are created at t_{now} and they are inserted to partition $P_{1,0}$ (sub-partition $P_{1,0}^{O_L}$).

Deletion events. When a deletion event arrives for record r carrying an $s.end$, i.e., an existing live interval s is terminated and becomes dead, we need to remove s from the live sub-partitions of *te*-HINT and add it to the appropriate dead partitions. For this, we use $\mathcal{H}_{r.id \rightarrow start}$ to retrieve $s.start$, using $r.id$, and we run the insertion algorithm of HINT for $s' = [s.start, t_H)$ to identify the partitions wherein s' appears and remove s' from the corresponding live sub-partitions. Subsequently, we use the insertion algorithm again to add $s = [s.start, s.end)$ to the relevant dead sub-partitions. Note that some of the partitions identified by the deletion algorithm may differ from those found by the insertion algorithm, because $s \neq s'$. As an example, assume that at time t_{now} shown in Figure 6.1(b), a deletion event for live interval s_2 arrives, i.e., the record version corresponding to s_2 is deleted from the indexed table T. After finding $s_2.start$ using $\mathcal{H}_{r.id \rightarrow start}$, the partitions ($P^{O_L}_{1,0}$) where s_2 is stored as live are identified using interval $[s.start, t_H)$ and s_2 is removed from them, and, finally, s_2 becomes $[s_2.start, t_{now})$ and is re-inserted to *te*-HINT as dead (i.e., to partition $P^{O_D}_{1,0}$).

Domain Extension. *te*-HINT is initialized to have a single level (0) which includes a single partition $P_{0,0}$. The timespan $[0, t_H)$ of the partition is small (e.g., one hour) and depends on the application. In both insert and delete events, it may happen that the current time point t_{now} when the update takes place is beyond the current horizon t_H of *te*-HINT. Such an update triggers the *extension* of the (time) domain that *te*-HINT covers. The easiest way to accommodate this extension is to double the domain (and the horizon t_H), by adding one more level to *te*-HINT (and repeat as necessary). Specifically, we add a new level 0 to the index and add 1 to the identifiers of existing levels (i.e., previous level 0 becomes level 1, level 1 becomes level 2, etc.). This does not affect the identifiers and contents of existing partitions at each level ℓ, but doubles the number of possible partitions at ℓ. Subsequently, we add all live intervals from all partitions as *live replicas* to partition $P_{1,1}$, except from those in old partition $P_{0,0}$ which are moved to the new $P_{0,0}$. By this, we minimize the replication of live intervals and also minimize the necessary updates when new events arrive. Essentially, live intervals are moved *only* when there is a domain extension. Continuing the previous example, assume that a new live interval s_4 is created at t_{now} of Figure 6.1(c). Since t_{now} is greater than or equal to t_H, as per the previous state of *te*-HINT (Figure 6.1(b)), t_H is doubled, one more level is added to *te*-HINT, and the current partitions are renamed (i.e., previous $P_{0,0}$ now becomes $P_{1,0}$, etc.), without any change in their contents. Existing live interval s_3 is added to the new partition $P^{R_L}_{1,1}$. The new interval s_4 is added using the insertion algorithm to $P^{O_L}_{1,1}$.

6.2 The LIT Hybrid Index

Capitalizing on the original HINT, *te*-HINT will deliver excellent performance on pure time-travel queries, as shown in [40]. But, *te*-HINT will suffer from slow updates, mainly due to the insertion (and transfer) of intervals to (and beween) multiple partitions when record versions are initiated (terminated). In view of this shortcoming, we design a *hybrid* index, termed LIT, which decouples the indexing of live and dead versions. For now, we describe LIT for pure time-travel queries. Its extension

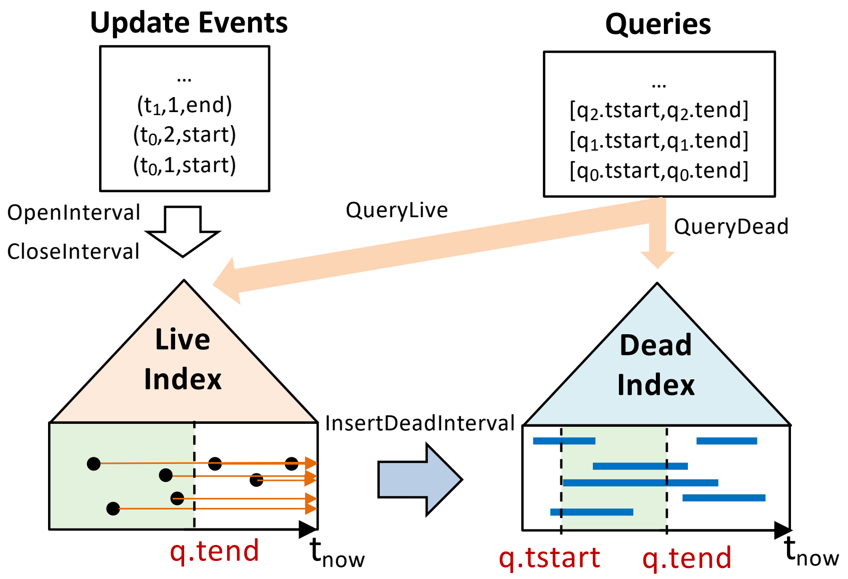


Figure 6.2: Overview of LIT

for range time-travel queries will be discussed in Section 6.3.

Overview of LIT Figure 6.2 shows an overview of LIT, which comprises two components; a LiveIndex denoted by $\mathcal{I}_L$, storing all current record versions (indexed by their $start$ timepoint) and a DeadIndex, denoted by $\mathcal{I}_D$, for the dead (i.e., past) record versions (indexed by their validity intervals). Both components are dynamic, albeit handling different updates. The stream of updates to the indexed table T is consumed by the LiveIndex $\mathcal{I}_L$. Specifically, when a new record version is created (i.e., an insertion to T), the start point $s.start = t_{now}$ of its validity interval is inserted to $\mathcal{I}_L$; this event type has no impact on the DeadIndex $\mathcal{I}_D$. On the other hand, when a record version "dies" (i.e., a deletion from T), the corresponding entry is removed from $\mathcal{I}_L$ and an entry is inserted to $\mathcal{I}_D$ for the dead record version. As already mentioned, record updates are treated by terminating (i.e., "deleting") the current (live) version of the record and inserting a new version.

To evaluate a pure time-travel query $q = [q.tstart, q.tend]$ both $\mathcal{I}_L$, $\mathcal{I}_D$ need to be probed. As the two components index disjoint sets of record versions, these probing tasks are completely independent. Specifically, we probe the LiveIndex $\mathcal{I}_L$ using only $q.tend$; every live record that started before $q.tend$ is guaranteed to be part of the query result. In contrast, the DeadIndex evaluates a typical interval range query to find all dead record versions with a validity interval that overlaps q. In what follows, we elaborate on the internals of the LIT components $\mathcal{I}_L$ and $\mathcal{I}_D$, and describe their key operations.

6.2.1 The LiveIndex Component

The LiveIndex $\mathcal{I}_L$ offers three key operations. Specifically, $\mathcal{I}_L$ is updated to index a new live record (Function OpenInterval) or updated to un-index a record version that just died (Function CloseInterval). $\mathcal{I}_L$ also evaluates pure time-travel queries (Function QueryLive). To efficiently implement these functions, $\mathcal{I}_L$ defines an internal identifier $r.num$ for each live record version r in it. The num identifier is a serial number that captures the order in which the version $start$ timepoints were read from the input stream of updates; num is used to (1) locate a live version to be deleted from $\mathcal{I}_L$ when a delete event arrives for it, and (2) define an implicit order of the live versions based on their $start$ points, used to index them in $\mathcal{I}_L$. LiveIndex also maintains an auxiliary hash table $\mathcal{H}_{r.id \to num}$, which returns the internal num id, for the live version of a given record id.

Data structures

We discuss three alternative data structures for LiveIndex, aiming at both fast updates and efficient time-travel queries. We experimentally compare them in Sec. 6.5.2.

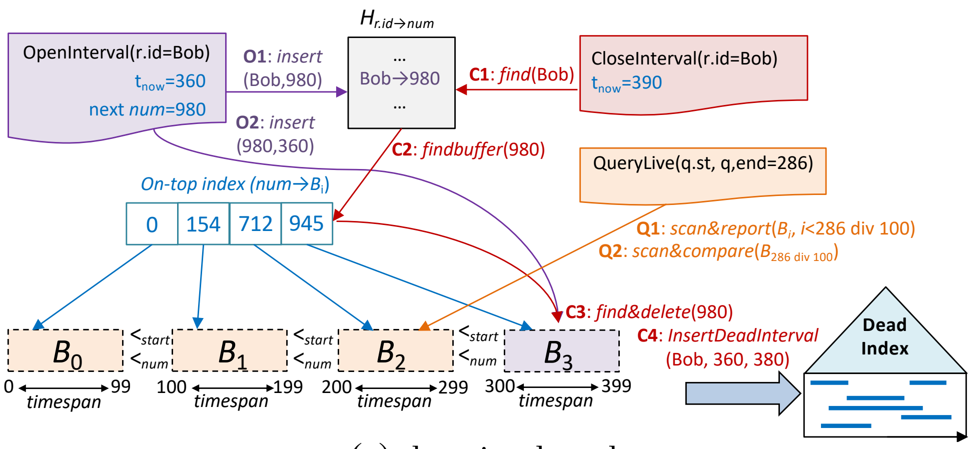
Array. The first alternative is to use an *append-only array* to index live records in sequential fashion. Updates can be efficiently handled in constant time, as follows. Function OpenInterval simply appends an entry at the end of the array for a new live record version, while CloseInterval, drops a tombstone on the existing entry for a newly closed record version. This entry can be directly accessed using the num of the record, which is obtained by probing the record id against $\mathcal{H}_{r.id \to num}$. To answer queries, the QueryLive function scans the dynamic array from its first entry, comparing the $start$ of every live record to $q.tend$ while ignoring the tombstones. By construction, the dynamic array stores the live records sorted by their num, which means that the records are also implicitly sorted by their $start$, in increasing order. Hence, QueryLive terminates the scan when the first record that started after $q.tend$ is accessed.

Search tree. A second alternative data structure for the LiveIndex $\mathcal{I}_L$ is a search tree (e.g., a B^+-tree), using num as the search key. With such a search tree in place, we no longer need to lazy-update $\mathcal{I}_L$ when a record version dies. Instead, CloseInterval probes the tree using the num identifier of the record (obtained from $\mathcal{H}_{r.id \to num}$), and then directly removes the corresponding entry. As a search tree typically supports scanning its entries in the search key order, to answer a $[q.tstart, q.tend]$ query, we scan and report from the first entry until we find the first that has its $start$ after $q.tend$.

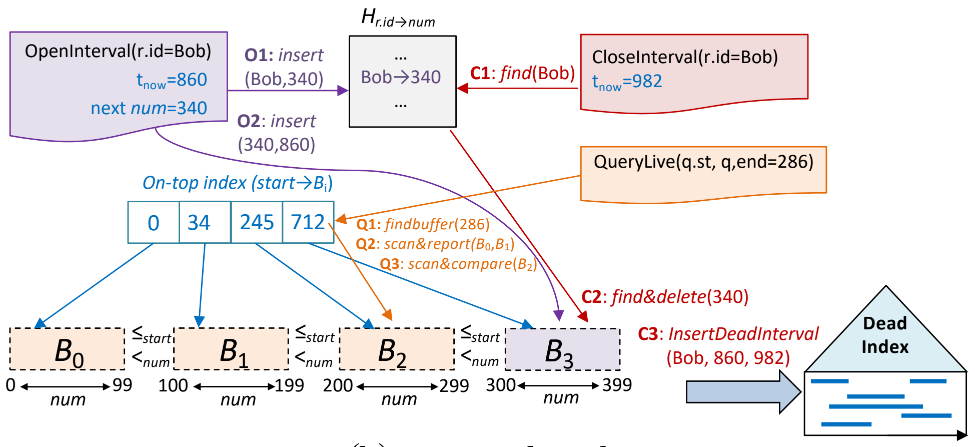
Enhanced hashmap. In terms of updating (Functions OpenInterval and CloseInterval), we generally expect the sorted array to outperform the search tree, due to its simplicity. Querying efficiency depends on the characteristics of the input stream; update-heavy workloads create a large amount of tombstones to the array, rendering it slower than the search tree. In view of the above, we should consider a data structure, which will exhibit competitive update time to the array and have lower query time. To this end, we suggest using an *enhanced* hash table, similar to the Gapless hashmap proposed in [26] or the java.util.LinkedHashMap in Java. Such structures can handle insertions and deletions using num in constant time (typical for hash tables), but also offer scan time linear to the number of contained entries, which facilitates fast query processing. In particular, the Gapless hashmap uses a contiguous memory area to store the elements. Insertions append new elements at the end of this area, while deletions are handled by swapping the deleted element with the last one and reducing the array size by one. Scanning is fast as it steps through the contiguous storage area sequentially. Different to both the array and the search tree, the hashmap does not maintain the entries sorted by their num, and therefore, a full scan is required to answer time-travel queries.

Temporal partitioning of LiveIndex

Given a query, a LiveIndex implemented by any of the data structures in Section 6.2.1 would need to conduct comparisons for a large number of live versions (independently of the underlying data structure), since there is no way to directly output versions guaranteed to start before $q.tend$. In view of this, we propose a *temporal partitioning* of the LiveIndex to boost time-travel queries. The key idea is to maintain $\mathcal{I}_L$ as a *chain of temporal partitions* or simply *buffers*, instead of a single one, such that all num's in a buffer are smaller than all num's in the next buffer. Hence, the $start$ points of live record versions in a buffer are smaller than or equal to the $start$ points of live versions in the next buffer. For each query, only the buffers that may contain results are accessed and even more importantly, comparisons are conducted only for the last buffer. This partitioning of the LiveIndex $\mathcal{I}_L$ is orthogonal to the data structure used



(a) duration-based



(b) capacity-based

Figure 6.3: LiveIndex: partitioning

for each buffer.

Duration-based partitioning. An intuitive partitioning approach for $\mathcal{I}_L$ is to consider a *duration constraint* D_L. Under this, $\mathcal{I}_L$ essentially resembles a uniform 1D-grid of equi-sized partitions, one for each buffer. A buffer B_i contains the live entries that started inside the $[i \cdot D_L, (i+1) \cdot D_L)$ range of time units. Given a $[q.tstart, q.tend]$ time-travel query, we first determine the bucket B_{end} that contains the $q.tend$ timestamp; this can be done in constant time by a simple $\lfloor q.tend/D_L \rfloor$ division. The records inside the buffers before B_{end} can be directly reported as results; by construction of the LiveIndex, these records started before $q.tend$. In contrast, comparisons against $q.tend$ are required for the live records inside the last B_{end}, i.e., QueryLive handles B_{end} as if the LiveIndex comprised a single buffer. Regarding updates, inserting a new live record version to $\mathcal{I}_L$ (Function OpenInterval) is not significantly affected by the above partitioning, as the new entry will be added to the last buffer, i.e., the one containing the most fresh records; extra action is required when D_L time units have already past and a new buffer needs to be created first. CloseInterval is more challenging, as we need to fast determine the buffer which contains the $start$ of the dying record version. For this purpose, we define an auxiliary, lightweight structure on top of the buffers. This structure stores a $\langle num, ptr \rangle$ entry for each buffer B of $\mathcal{I}_L$, where num is the lowest internal identifier of a live record version inside B and ptr is a pointer to directly access B in the chain. Recall at this point, that LiveIndex is organized by num and so is its on-top structure, by construction. When a version of record $r.id$ dies, CloseInterval finds its num using $\mathcal{H}_{r.id \rightarrow num}$, then binary-searches the on-top structure using $r.num$ and, lastly, follows the buffer pointer to locate the entry for num inside the corresponding buffer B. After deleting the entry from $\mathcal{I}_L$, CloseInterval, forwards the dead version for insertion to $\mathcal{I}_D$. OpenInterval may update the on-top structure when the last buffer is full and a new is created. Figure 6.3(a) exemplifies a *duration-based* partitioned LiveIndex, with the necessary steps taken for each of the OpenInterval, CloseInterval, and QueryLive operations.

Capacity-based partitioning. Duration-based partitioning may define unbalanced buffers with respect to the number of contained entries, rendering unbalanced query costs. An alternative partitioning approach that results in balanced partitions is to use a capacity constraint C_L, allowing each buffer to hold at most C_L entries. [1] Different to the duration-based partitioning discussed above, capacity-based partitioning can directly access the needed buffers during both types of updates. For OpenInterval, we simply append the new live record version at the last buffer, while for CloseInterval, a simple num/C_L division exactly determines which buffer B contains the recently deceased version. Note that if the last buffer is already full, OpenInterval will create a new buffer B_{new} after the last one and simply append the new live version in B_{new}.

On the other hand, it is no longer possible to directly determine the B_{end} buffer for a $[q.tstart, q.tend]$ query. In view of this, we define an on-top structure, which stores a $\langle st, ptr \rangle$ entry for each buffer B of the LiveIndex, where st is the lowest $start$ timepoint of a record version inside B and ptr is a pointer to directly access B. Note that the on-top search structure is by construction sorted by version $start$ and that it may contain multiple entries for the same $start$. Hence, given a $[q.tstart, q.tend]$ query, QueryLive first binary-searches the on-top structure to identify the *first* buffer that could contain $q.tend$ and sets this as B_{end}. With B_{end}, the function proceeds similarly to the duration-based LiveIndex, by directly reporting records inside every buffer before B_{end} and conducting comparisons against $q.tend$ for B_{end}. Lastly, besides up-

[1]For array structure, tombstones are *not* excluded when counting the contained records.

dating buffers, OpenInterval and CloseInterval also update accordingly the on-top structure. Figure 6.3(b) illustrates a detailed example of the *capacity-based* partitioning of LiveIndex and operations on it.

Optimizations

As the timeline evolves and live records die, buffers may become under-utilized or even completely empty. To deal with this issue, reorganization can be employed for both types of partitioning. For the duration-based LiveIndex, such a sparsity issue is expected to especially occur in the first (early) buffers. Hence, we could merge adjacent sparse buffers into one and accordingly update also the on-top structure.[2] To answer time-travel queries, an extra auxiliary structure is now needed to capture the time-range covered by this new buffer, as the $q.tend/D_L$ division can only work for un-merged buffers. Intuitively, a second on-top structure maintaining the lowest $start$ inside a buffer will allow us to deal with several rounds of buffer merging. For the capacity-based LiveIndex, one solution would be to define a lower-bound for the capacity of a buffer. When the capacity of a buffer drops below e.g., 50% of C_L, we mark the buffer and merge it with either its predecessor or its follower (if one of them is also marked), and then update accordingly the on-top structure. Finally, similar to the duration-based LiveIndex, a new on-top structure is again required, as the num/C_L division no longer works. This new structure will hold the lowest num inside a buffer, and will be binary searched by CloseInterval.

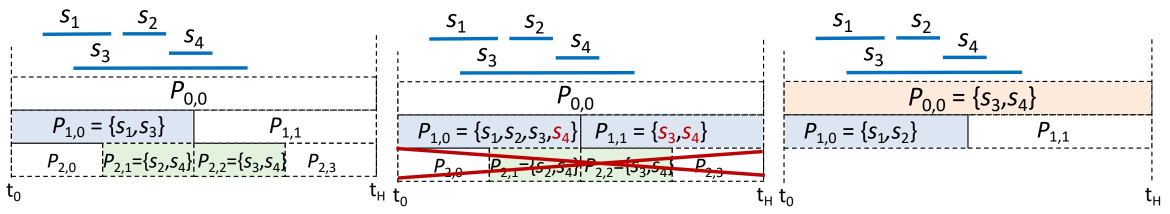
6.2.2 The DeadIndex Component

We now turn our focus on indexing dead record versions. Recall that these versions were evicted from the LiveIndex $\mathcal{I}_L$ by the CloseInterval function, after their end was read from the input stream. The DeadIndex $\mathcal{I}_D$ offers two key operations. Specifically, (1) $\mathcal{I}_D$ is updated to index a new dead record version (Function InsertDeadInterval) and (2) it evaluates time-travel queries (Function QueryDead). As the timeline evolves and new dead versions are added to $\mathcal{I}_D$, its domain grows. Under this, a straightforward solution for indexing dead record versions is the 2D point transformation approach from [1] as discussed in Section 3.1, where a 2D spatial index such as the R-tree, can adapt to the growing domain.

[2]The number of buffers to be merged can be seen as a tunable system parameter.

An alternative solution is to modify the state-of-the-art interval index HINT [40] to adapt to a growing domain. Section 6.1.2 already discusses this in the context of *te*-HINT. Implementing domain extension for a HINT DeadIndex is simpler, because we do not have to deal with transfers of live intervals between buckets as in *te*-HINT. Instead, we only have to add one more level and double the horizon t_H, as soon as we cannot accommodate a newly inserted interval s having at least one of its endpoints after t_H. As in *te*-HINT, after the expansion operation, the existing partitions are renamed to reflect their new level, but their contents remain intact.

Increasing the number of levels in a HINT that implements $\mathcal{I}_D$ to a very large number may negatively affect its search performance and size, as there could be far too many partitions for the number of indexed intervals [40]. A naïve approach to reduce the number of HINT levels by one is to construct a new HINT with one less level and insert all intervals in it. We propose a more efficient algorithm for deleting the lowest level of HINT, which progressively moves intervals from the deleted level to an appropriate partition above, while maintaining the HINT property (i.e., each interval s should be assigned at the smallest set of partitions from all level that define s). Each interval at level m (to be deleted) is stored in at most two level-m partitions. Intervals that begin and end in exactly one partition $P_{m,i}$ are directly moved to $P_{m-1,i \div 2}$ and no further action is needed. This is the case of s_2 in Figure 6.4(a) which is moved to $P_{1,0}$ in Figure 6.4(b). Intervals that begin in a $P_{m,i}$, for an odd i, are *temporarily moved* to $P_{m-1,i \div 2}$; the same holds for intervals that end in a $P_{m,i}$, for an even i. For instance, s_3 in Figure 6.4(a) is temporarily moved to partition $P_{1,1}$ because it ends in $P_{2,2}$, while s_4 is temporarily moved to both $P_{1,0}$ and $P_{1,1}$ (see Figure 6.4(b)). Temporary partitions $P_{m-1,j}$ at each level $\ell < m$ for an even j are set-intersected with the next partition at the same level holding replicas, at the potential of moving intervals to the previous level $\ell - 1$ as finalized or temporary. Symmetrically, temporary partitions $P_{\ell,j}$ at level ℓ for an odd j are set-intersected with the previous partition $P_{\ell,j-1}$. While there are temporary partitions at each level, intervals may propagate upwards until their correct partition is found. For instance, intervals s_3 and s_4, which, after the deletion of level 2, were stored in (temporary) partitions $P_{1,0}$ and $P_{1,1}$ at level 1 are eventually propagated at $P_{0,0}$ of level 0, as shown in the final HINT at Figure 6.4(c). A pseudocode of the drop level algorithm is skipped due to space constraints. Note that the same method can be used to delete the last level of *te*-HINT.



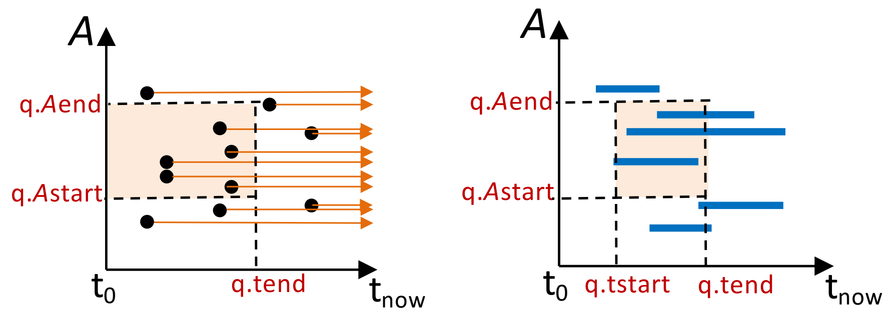
(a) before level drop (b) level m drop (c) propagation and final HINT

Figure 6.4: Steps of dropping last level m of HINT ($m = 2$)

6.3 Indexing Record Attributes

We now discuss how to modify LIT and index record versions on a specific attribute A for range time-travel queries, where not only a timepoint/range is specified but also a selection predicate on A. We denote a LIT that indexes an attribute A (besides time) by a-LIT.

Before describing a-LIT we discuss the requirements of a LiveIndex and a DeadIndex in the presence of the attribute A. Figure 6.5 illustrates the information that should be stored about live and dead record versions. As shown in Figure 6.5(a), to be able to answer range time-travel queries against LiveIndex, we need for each live version its *start* point and its A-value. So, the live version is a 2D point in the time-A space. A range timer-travel query can then be modeled as a rectangular range $\{[t_0, q.end], [q.Astart, q.Aend]\}$ in the time-A space. Regarding the DeadIndex, we need for each dead version its *start*, *end* and its A-value. Figure 6.5(b) illustrates some dead versions in the time-A space and a range time-travel query, which is modeled as a 2D rectangle, defined by the query bounds.



(a) Live space (b) Dead space

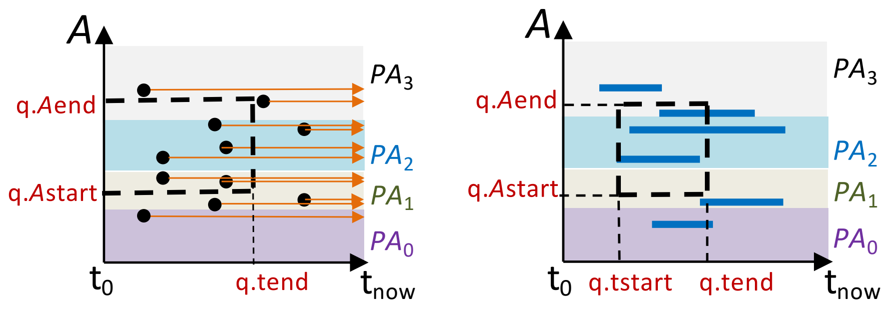
Figure 6.5: Live and Dead space and queries

6.3.1 The LiveIndex Component

The LiveIndex of a-LIT should index the $start$ timepoints of the current record versions and their values on A simultaneously.

2D space index. A natural approach to do so would be to use a native index for 2D points (e.g., kd-tree, quadtree, R-tree). Besides the 2D-space index, we also need an auxiliary structure $\mathcal{H}_{r.id \to (start,A)}$ that maps record ids to the $start$ points of their live versions and their A values. Otherwise, it would not be possible to find and remove an indexed point from the 2D index, when the corresponding version dies (i.e., CloseInterval). Hence, the OpenInterval operation inserts the $(start = t_{now}, A)$ entry of a new live version to both the 2D index and $\mathcal{H}_{r.id \to (start,A)}$. Operation CloseInterval uses $\mathcal{H}_{r.id \to (start,A)}$ to find the coordinates of the ending version in the 2D index, searches and removes it, and relays the dead record version to DeadIndex. Finally, QueryLive issues a 2D query to the 2D index to retrieve the qualifying live versions.

Use multiple pure time indices. Another indexing approach is to divide the domain of A into partitions (e.g., equi-width) and develop a LiveIndex as described in Section 6.2.1 for each partition. The data structures and temporal partitioning methods are defined separately for each partition. The only difference is that the mapping mechanism $\mathcal{H}_{r.id \to num}$ of record ids to num values should also capture the A-partition identifier wherein a live version is located. By this, CloseInterval can identify and delete a live version from the correct A-partition of the LiveIndex. Figure 6.6(a) illustrates an A-partitioning of the live data space into four divisions (PA_0 to PA_3). For each of them, we can define a pure temporal LiveIndex, as described in Section 6.2.1. Given a range time-travel query, we use the selection predicate on A to identify the partitions that overlap with the query range in the A-domain (i.e., PA_1, PA_2, and PA_3 in Fig. 6.6(a)). If a partition is entirely covered by the A-range of the query (e.g., partition PA_2), we evaluate the temporal part of the query, as described in Sec. 6.2.1. Otherwise (e.g., in PA_1 and PA_3), for each result obtained by the LiveIndex of the partition, we verify the A-predicate of the query. This verification is applied for at most two A-partitions containing the query boundaries. Updates on this A-partitioning approach are expected to be faster than updates on a 2D index, due to the fast hashing mechanisms it incorporates.



(a) Live space A-partitioning (b) Dead space A-partitioning

Figure 6.6: Live and Dead space A-partitioning

6.3.2 The DeadIndex Component

Now we turn to DeadIndex options for a-LIT. Like before, we can follow either a pure geometric approach or apply an A-partitioning technique to take advantage of the efficiency of pure time indices.

3D index. A straightforward approach is to index the line segments of the dead space (see Fig. 6.5(b)) directly by a native 2D index for geometric objects (e.g., an R-tree). However, such a method is not expected to perform well because some record versions in temporal databases are *long-lived* and correspond to very long segments that require large node MBRs, rendering the index inefficient. A more effective approach is to model each dead version as a 3D point $(s.start, s.end, r.A)$ in the (time, time, A) space, and index these points using a 3D index (e.g., a 3D R-tree). Figure 3.4(a) shows how this can be done for pure time intervals; the idea is to add one more dimension for A. Every query in this 3D space is then modeled as a $([0, q.tend], [q.tstart, t_{now}], [q.Astart, q.Aend])$ 3D box.

Use multiple pure time indices. Similar to the case of LiveIndex, we may also partition the domain of A to define a number of partitions, as shown in Figure 6.6(b). For each partition (e.g., PA_0 to PA_3), we use an optimized interval index, such as the modified HINT to support domain extension, discussed in Section 6.2.2. Given a range time-travel query, we first identify the A-partitions that overlap with the query A-range (e.g., PA_1, PA_2, PA_3) and then evaluate a pure time-travel query in each such partition, verifying the A-predicate against its results if necessary (e.g., in PA_1 and PA_3).

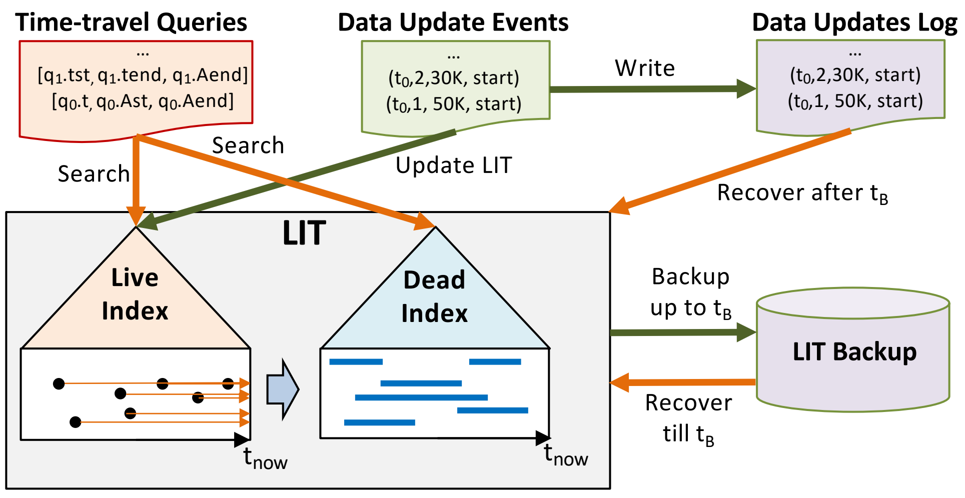


Figure 6.7: Persistence and recovery of LIT

6.4 Persistence and Recovery

LIT is a main-memory index that facilitates real-time analytics, high-performance querying, and handling large volumes of rapidly changing temporal data. However, since main memory is volatile, we should ensure durability and recoverability, after power or system failures. Figure 6.7 illustrates how LIT is integrated into a temporal database system, to support fault tolerance and recovery. For this, each update event is written to a log file. In addition, a backup of LIT is taken periodically and written to the hard disk for persistence and faster recovery. The backup is merely a dump of the main memory data structures for LiveIndex and DeadIndex. Assuming that the last checkpoint where the last backup has been taken is t_B, to recover LIT at a time $t_{now} > t_B$ (e.g., due to a power failure at that time), we first load the backups of LiveIndex and DeadIndex in main memory, then find the first event after t_B in the log file, and finally ingest all events after t_B to evolve LiveIndex and DeadIndex to their current state at t_{now}. Since all states up to t_B are captured by the LIT backup, we can even “cleanup” the log file by removing all entries up to t_B, to avoid searching it.

Table 6.1: Characteristics of tested datasets

		TAXIS-F	TAXIS-P	BIKES	FLIGHTS	WILDFIRES	BOOKS
	Cardinality	169290307	169290307	101472950	61328124	778410	2050707
temporal	**Min duration**	1 min	1 min	1 min	5 min	1 min	1 hour
	Max duration	5 hours	5 hours	7.5 months	12 hours	4 months	1 year
	Avg. duration	12 mins	12 mins	16 mins	2.5 hours	28 hours	67 days
search-key	**Description**	trip fare [USD]	passengers count	rider’s birth year	departure delay [secs]	number of books lent	fire extent [acres]
	Type	real	integer	integer	real	integer	real
	Value range	$[2.5, 235.5]$	$[1, 6]$	$[1940, 2005]$	$[0, 233400]$	$[1, 38]$	$[0.0001, 606945]$
	Distribution	normal	zipfian	normal	zipfian	zipfian	zipfian

Table 6.2: Query extents; default values in bold

input stream	query extent	
	temporal	search-key
TAXIS-F	1, 6, **12**, 18, 24 [hours]	3, 5, **10**, 30, 50 [dollars]
TAXIS-P	1, 6, **12**, 18, 24 [hours]	1, 2, **3**, 4, 5 [passengers]
BIKES	1, 6, **12**, 18, 24 [hours]	10, 20, **30**, 40, 50 [years]
FLIGHTS	1, 2, **3**, 4, 5 [days]	5, 10, **30**, 60, 120 [mins]
WILDFIRES	1, 7, **14**, 21, 30 [days]	10, 50, **100**, 500, 1000 [acres]
BOOKS	1, 7, **14**, 21, 30 [days]	5, 10, **15**, 20, 25 [books]

6.5 Experimental Analysis

We last present the results of our experimental analysis. All indices were implemented in C++, compiled using gcc (v9.4.0) with flags -O3, -mavx, and -march=native. The tests ran on a AMD Ryzen 9 CPU, clocked at 3.5GHz with 64 GB of RAM, running Ubuntu 20.04.

6.5.1 Setup

Datasets. We experimented with 6 real-world temporal datasets which also include a search-key attribute A; Table 6.1 summarizes their characteristics. TAXIS-F and TAXIS-P contain the pick-up and drop-off timepoints of taxi trips (same intervals in both datasets) in NYC from 2009.[3] In TAXIS-F, A is the paid fare, and in TAXIS-P A is the number of passengers. BIKES[4] contains the pick-up and drop-off timepoints of bike rides in NYC from 2014 to 2021; the search-key A is the birth year of the rider. FLIGHTS[5] contains the take-off and landing timepoints of flights recorded by the US Transportation Department from 2013 to 2022, and the occurred departure delay. BOOKS [6] contains the periods of time when books were lent out by Aarhus libraries in 2013, and the number of books during each period. WILDFIRES[7] specfies when a fire was discovered and when declared contained/controlled. As search-key A, we use an estimate of the area burnt. BOOKS, WILDFIRES include objects with long validity intervals, while in TAXIS, BIKES time intervals are extremely short; FLIGHTS lies in

[3]https://www1.nyc.gov/site/tlc/index.page

[4]https://citibikenyc.com/system-data

[5]https://www.bts.gov

[6]https://www.odaa.dk

[7]https://www.kaggle.com/datasets/rtatman/188-million-us-wildfires

the middle of the spectrum. As search-key, we consider both real and integer values; A's domain varies from extremely small (TAXIS-P) to extremely large (WILDFIRES). Last, the values of A follow either a normal or a Zipfian distribution.

Input streams. We created an event stream (workload) out of every dataset, by splitting each interval to an insert and a deletion event, and interleaving 10K queries. Queries are positioned uniformly inside the active timeline, i.e., the period between the $start$ of the very first interval until current t_{now}. The nature of the created streams varies from extremely update-heavy for TAXIS, BIKES and FLIGHTS with a 34000/1, 20000/1 and 13000/1 ratio of updates over queries, respectively, to moderate for BOOKS and WILDFIRES, with a 410/1 and 156/1 ratio, respectively. We considered two types of query extents; for pure time-travel queries, the extent of the $[q.tstart, q.tend]$ interval while for range time-travel queries, additionally the extent of the $[q.Astart, q.Aend]$ range. Table 6.2 lists the values for the query extents; the defaults are in bold. In each test, we measure the update time (for some indices, broken down to insert and delete time) and the query time.

6.5.2 Pure time-travel Queries

We start our evaluation with pure time-travel queries. As we ignore the search-key A, we consider a single TAXIS stream.

Tuning LIT

We first investigate the most efficient structure and partitioning for the LiveIndex, and the most

LiveIndex: data structure. We implemented the alternative structures from Section 6.2.1; STL C++ `vector` class was used for the *append-only array*, STL C++ `ordered_map` class (Red-Black tree) for the *search tree* and the Gapless hashmap from [26] for the *enhanced hashmap*.[8] Table 6.3 summarizes the results of our tests; for the interest of space, we report only BOOKS and TAXIS, which contain long and short intervals, respectively. The tests back up our intuition from Section 6.2.1. The append-only array exhibits the best (lowest) update times due to its simplicity. The enhanced hashmap however is always competitive, even for the update-heavy stream

[8]Source code was provided by the authors.

Table 6.3: LiveIndex for LIT; time in secs, default query extents
TAXIS

q **extent**	**Append-only array**				**Search tree**				**Enhanced hashmap**			
[hours]	ins	del	query	*total*	ins	del	query	*total*	ins	del	query	*total*
1	4.61	5.31	409	418.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
6	4.61	5.31	410	419.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
12	4.61	5.31	409	419.1	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
18	4.61	5.31	411	420.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43
24	4.61	5.31	412	421.9	18.6	29.3	0.001	47.90	5.10	7.32	0.011	12.43

BOOKS

q **extent**	**Append-only array**				**Search tree**				**Enhanced hashmap**			
[days]	ins	del	query	*total*	ins	del	query	*total*	ins	del	query	*total*
1	0.057	0.068	14.4	14.52	0.336	0.967	38.0	39.30	0.065	0.142	6.41	6.61
7	0.057	0.068	14.8	14.92	0.336	0.967	37.9	39.20	0.065	0.142	6.45	6.65
14	0.057	0.068	14.9	14.93	0.336	0.967	39.7	41.0	0.065	0.142	6.46	6.66
21	0.057	0.068	15.2	15.32	0.336	0.967	41.9	43.20	0.065	0.142	6.47	6.66
30	0.057	0.068	15.6	15.72	0.336	0.967	42.8	44.10	0.065	0.142	6.43	6.63

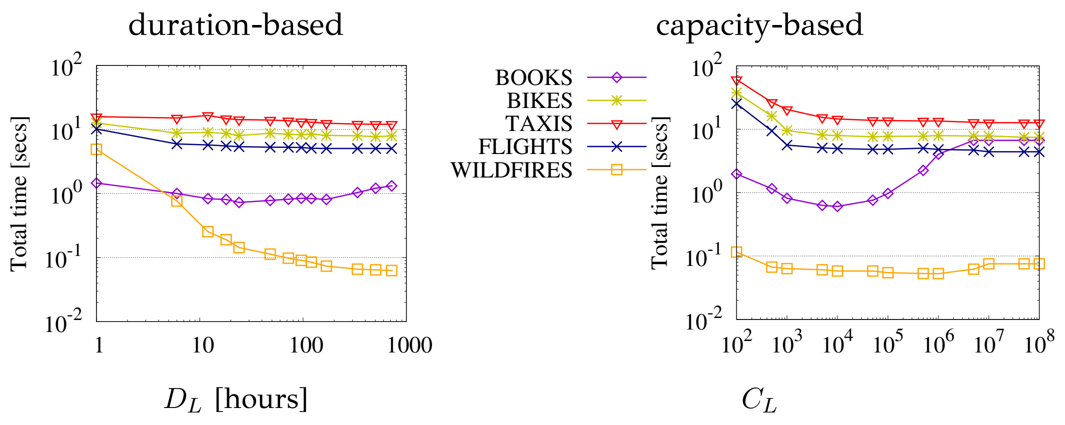


Figure 6.8: LiveIndex for LIT tuning; default query extent

of TAXIS. The search tree on the other hand is outperformed by an order of magnitude for both inserts and deletions. Regarding queries, the enhanced hashmap is the most robust structure; the efficiency of the other two is affected by the nature of the input stream and/or the length of the intervals. Update-heavy streams will incur a large number of tombstones and significantly slow down the append-only array, while long-lived intervals increase the size of LiveIndex and slow down the search tree. Overall, the enhanced hashmap offers the best trade-off between updates and queries, exhibiting always the lowest total time. For the rest of our experiments, we rely on the enhanced hashmap to store the LiveIndex.

LiveIndex: partitioning. We implemented both partitioning approaches from Sec-

Table 6.4: LiveIndex for LIT; in msecs; default extents

input	duration-based				capacity-based			
stream	insert	delete	query	*total*	insert	delete	query	*total*
TAXIS	4653	7462	4	12121	5252	7418	11.4	12681
BIKES	5667	2358	5	8030	3305	4140	4.3	7449
FLIGHTS	3059	2018	4	5083	1742	2653	11.1	4405
WILDFIRES	27	33	3	63	23	27	2.8	52.8
BOOKS	83	270	352	706	82.6	204	319	606

Table 6.5: DeadIndex for LIT; times in secs
TAXIS

query extent	2D R-tree [1]			HINT		
[hours]	insert	query	*total*	insert	query	*total*
1	69.7	3.21	72.9	8.43	0.28	8.71
6	69.7	15.5	85.2	8.43	1.54	9.97
12	69.7	29.8	99.5	8.43	2.96	11.4
18	69.7	44.3	114	8.43	3.39	11.8
24	69.7	59.2	128	8.43	6.20	14.6

BOOKS

query extent	2D R-tree [1]			HINT		
[days]	insert	query	*total*	insert	query	*total*
1	0.63	45.9	46.5	0.15	0.27	0.42
7	0.63	47.8	48.6	0.15	1.05	1.20
14	0.63	51.2	51.8	0.15	1.86	2.01
21	0.63	55.2	55.7	0.15	1.74	1.89
30	0.63	59.1	59.7	0.15	2.96	3.11

tion 6.2.1. To determine the best value for the duration constraint D_L and the capacity constraint C_L, we conducted the experiment in Figure 6.8 where the total time (update plus query time) is reported, while varying D_L and C_L. Note that as the value of both constraints increases, the number of LiveIndex buffers always drops. With the best observed values for each input stream in place, we compare the two approaches in Table 6.4, which also includes a runtime breakdown for each approach. We observe that the capacity-based partitioning always outperforms the duration-based by 10%, on average. For the rest of our analysis, as LiveIndex of LIT will use the capacity-based partitioning; also, based on Figure 6.8's experiment, we set $C_L = 10000$ for all streams.

DeadIndex. We compare HINT in the role of DeadIndex as discussed in Section 6.2.2,

Table 6.6: Pure time-travel queries: total update time [secs]

input stream	Timeline	*te*-HINT	LIT		
			LiveIndex	DeadIndex	total
TAXIS	12.3	1886	14.5	8.43	22.89
BIKES	10.4	357	7.93	5.13	13.06
FLIGHTS	4.08	526	4.68	3.01	7.69
WILDFIRES	0.05	0.38	0.07	0.04	0.11
BOOKS	0.19	349	0.49	0.14	0.63

Table 6.7: Pure time-travel queries: in secs; default extents

Component	**input stream**				
	TAXIS	BIKES	FLIGHTS	WILDFIRES	BOOKS
LIT: LiveIndex	0.157	0.005	0.011	0.001	0.371
LIT: DeadIndex	2.96	0.203	0.504	0.019	1.85

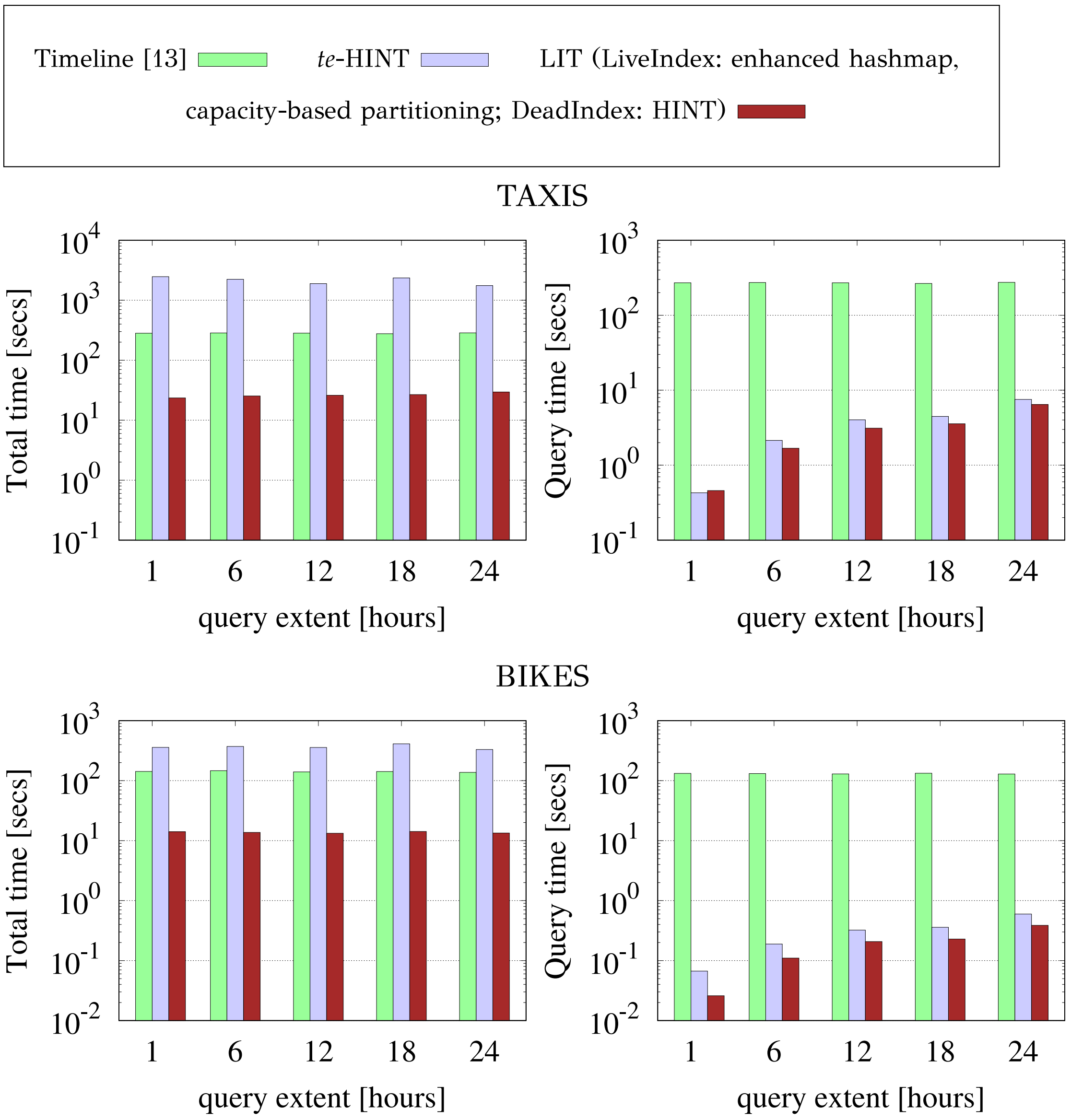
against the 2D transformation approach proposed in [1], powered by a 2D R-tree from the highly optimized Boost.Geometry library.[9] Table 6.5 reports the insert time and the query time for each DeadIndex approach, while varying the query extent. Due to lack of space, we show again only the numbers for BOOKS and TAXIS. HINT outperforms the 2D R-tree on computing pure time-travel queries by at least one order of magnitude (usually two orders), while for ingesting dead records, the 2D R-tree is competitive only in case of BOOKS, which contains significantly fewer updates than TAXIS. In contrast, for the update-heavy TAXIS, the 2D R-tree is an order of magnitude slower than HINT for indexing new dead records. In view of the above, LIT will use HINT as its DeadIndex component for the rest of our analysis.

LIT against the competition

We now compare the LIT hybrid index against *te*-HINT (Section 6.1) and the state-of-the-art Timeline index [13] for transactional DBs. Figure 6.9 (first row) reports the total time (updates and queries) for each index to ingest the input streams, while varying the query extent. Our tests clearly show that LIT is the most efficient index for all input streams, followed in almost all cases by the Timeline index, while *te*-HINT ranks last, with the exception of WILDFIRES. To better understand these results, the second row of the figure reports the accumulated time over the 10K queries of the

[9]Benchmark in [73] showed that Boost.Geometry (https://www.boost.org) R-tree implementations outperform the libspatialindex library (https://libspatialindex.org/).

stream and Table 6.6 reports the accumulated update time. The query costs of LIT and *te*-HINT are always lower compared to those of Timeline; *te*-HINT is competitive to LIT but in all cases slower. For updates, Table 6.6 shows the advantage of Timeline; recall from Chapter 3 that Timeline is designed for the support of fast updates in transaction-time DBs. Nevertheless, LIT is competitive to Timeline. Also, observe that the total updating cost is almost equally divided in between the LiveIndex and the DeadIndex. In contrast, *te*-HINT is orders of magnitude slower than LIT and Timeline in updates, mainly due to the high cost of moving intervals between partitions at different levels, as the timeline evolves and deletion events arrive. Overall, LIT offers the best tradeoff between updates and queries, resulting in the lowest total time, even for update-heavy streams such as TAXIS and BIKES. Lastly, we provide a breakdown



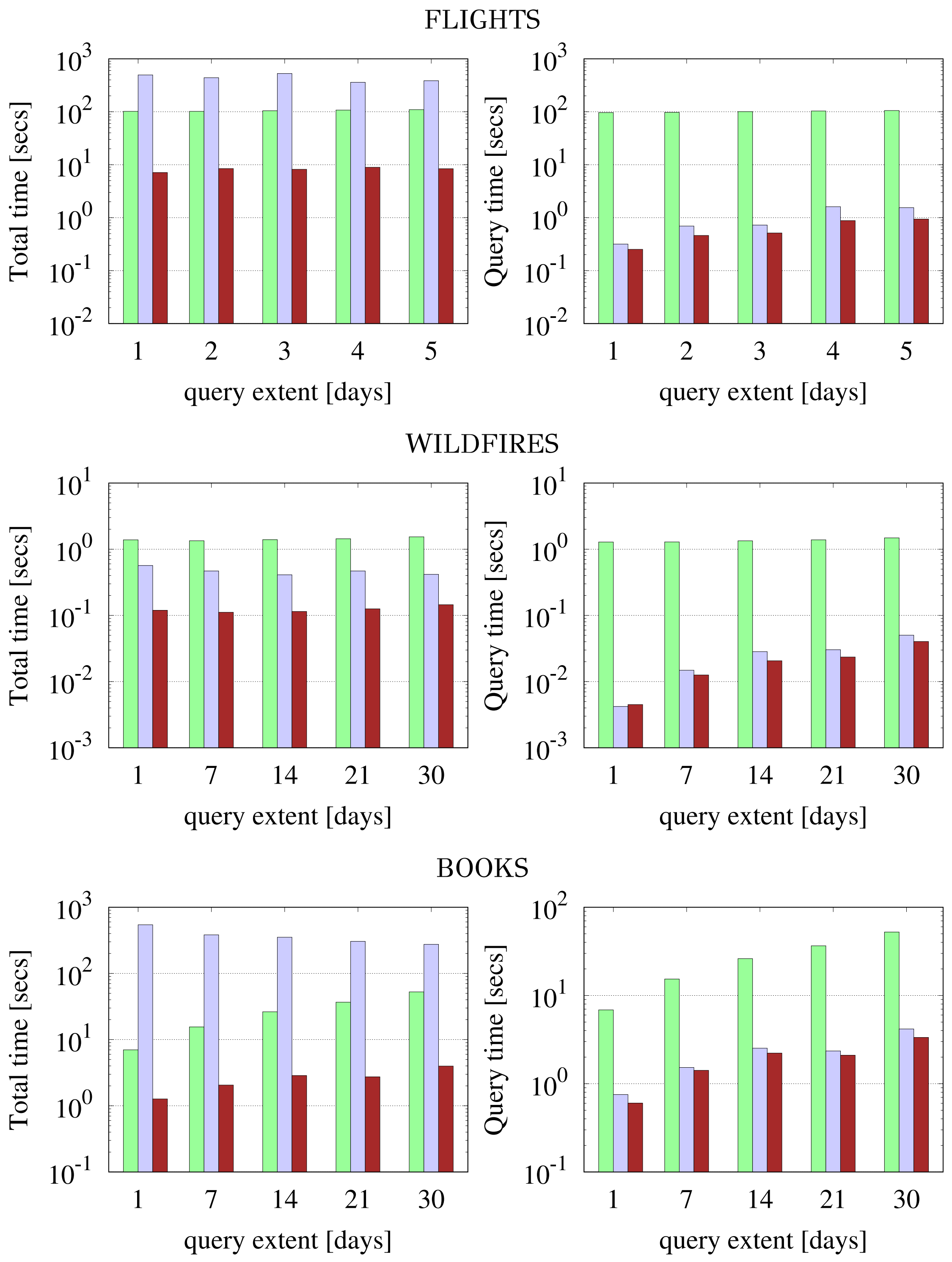


Figure 6.9: Pure time-travel queries

to the query time of LIT in Table 6.7.

6.5.3 Range time-travel Queries

We next switch gears and evaluate range time-travel queries, which include selections on the search-key A. For a-LIT, we considered an equi-width partitioning of the A domain in 6-7 partitions.

Tuning a-LIT

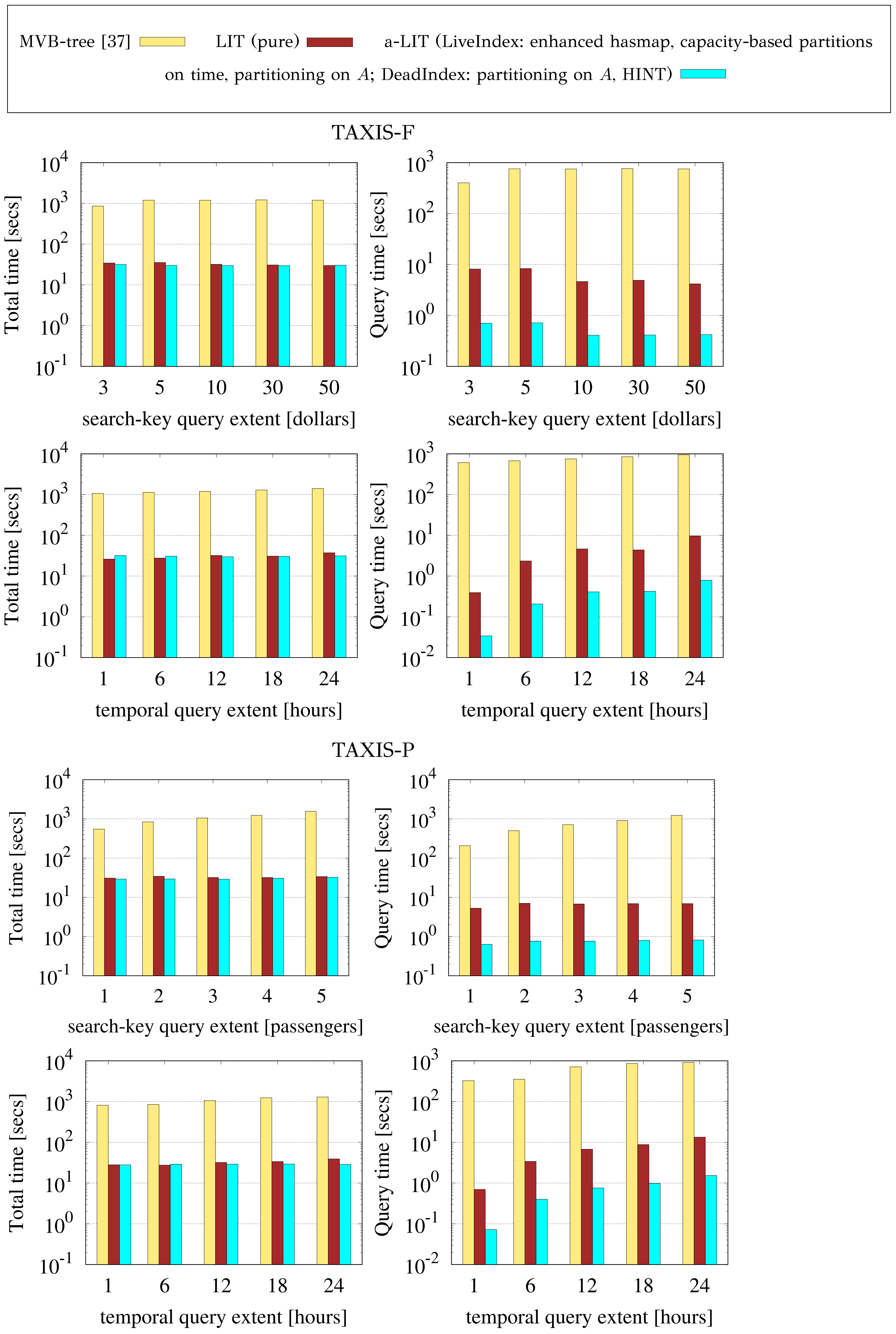
We first investigate the best setup for a-LIT.

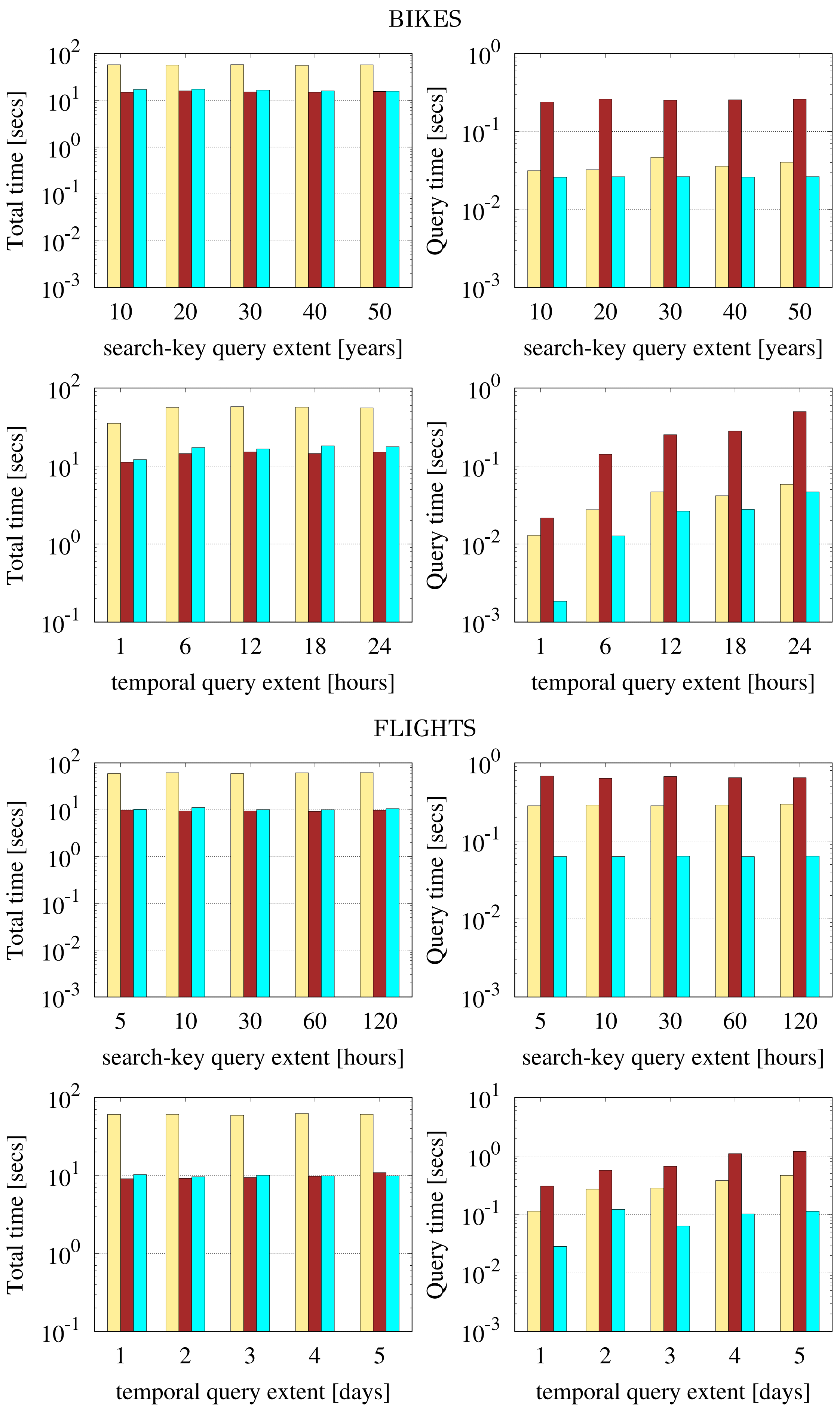
LiveIndex. We compared two alternative solutions for the LiveIndex, following our discussion in Section 6.3.1; a Boost 2D R-tree which directly indexes the $start$-A 2D space and a series of pure LiveIndices, one for each partition of the A domain. For the interest of space, we had to omit the results of this comparison. Our tests showed that the series of pure LiveIndices solution always outperforms the 2D R-tree LiveIndex, both for updates and queries, even for the update-heavy streams, i.e., TAXIS and BIKES.

DeadIndex. We implemented the two options discussed in Section 6.3.2, a 3D R-tree which directly indexes both the validity interval of a dead version and its search-key A, and a series of pure DeadIndices powered by HINT, one for each partition of the A-domain. Table 6.8 reports the total update (insert) and query time for each approach, while varying the search-key query extent; due to lack of space, we only report the case of the TAXIS-F and BOOKS streams. The table clearly shows the advantage of the multiple pure DeadIndices option in the role of the DeadIndex for a-LIT. The 3D R-tree DeadIndex is always slower both for updating (insertions of dead record versions) and querying. Especially in BOOKS, the performance gap rises to at least one order of magnitude because the 3D R-tree querying struggles with the long-live intervals.

a-LIT against competition

We compare a-LIT against two competitors. The first is a *time-first* baseline, which directly employs the pure LIT and does not index the search-key. To answer a range time-travel query q, this LIT (pure) first executes a pure time-travel query with


MVB-tree [37]
LIT (pure)
a-LIT (LiveIndex: enhanced hasmap, capacity-based partitions on time, partitioning on A; DeadIndex: partitioning on A, HINT)
TAXIS-F
Total time [secs]
Query time [secs]
search-key query extent [dollars]
temporal query extent [hours]
TAXIS-P
search-key query extent [passengers]
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
3
5
10
30
50
1
6
12
18
24
1
2
3
4
5


BIKES
Total time [secs]
Query time [secs]
search-key query extent [years]
temporal query extent [hours]
FLIGHTS
search-key query extent [hours]
temporal query extent [days]

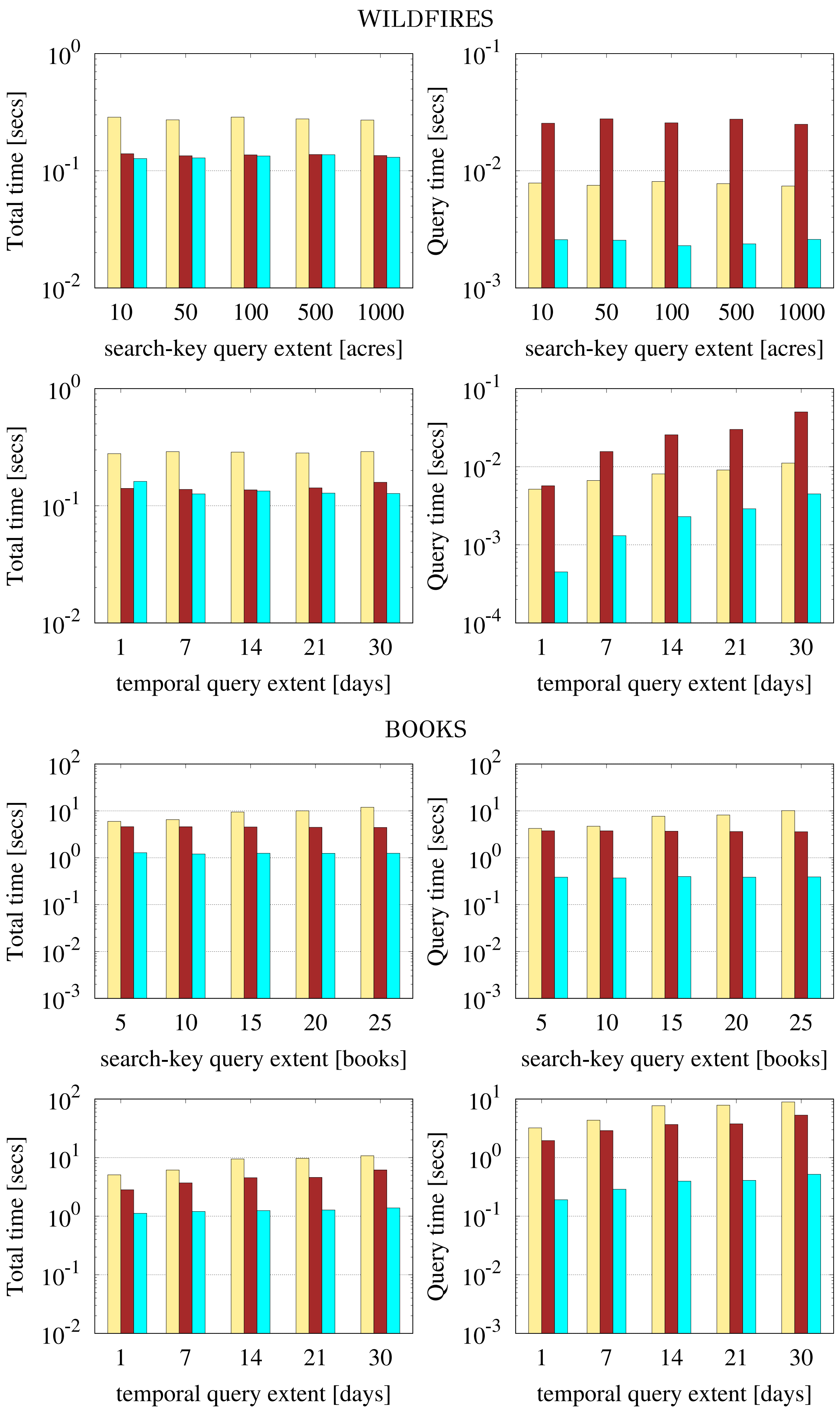


Figure 6.10: Range time-travel queries

Table 6.8: DeadIndex for a-LIT; times in secs
TAXIS-F

search-key	3D R-tree [1]			multiple HINTs		
query extent [dollars]	insert	query	*total*	insert	query	*total*
3	81.9	40.6	123	9.48	0.49	9.97
5	81.9	40.5	122	9.48	0.51	9.99
10	81.9	40.6	123	9.48	0.40	9.88
30	81.9	40.6	123	9.48	0.41	9.89
50	81.9	40.5	122	9.48	0.41	9.89

BOOKS

search-key	3D R-tree [1]			multiple HINTs		
query extent [books]	insert	query	*total*	insert	query	*total*
5	0.74	4.80	5.52	0.15	0.26	0.41
10	0.74	5.35	6.07	0.15	0.25	0.40
15	0.74	7.86	8.60	0.15	0.28	0.43
20	0.74	9.14	9.88	0.15	0.27	0.42
25	0.74	11.6	12.3	0.15	0.27	0.42

Table 6.9: Range time-travel queries: total update time [secs]

input stream	MVB-tree [37]	LIT (pure)	a-LIT
TAXIS-F(-P)	341	27.9	29.3
BIKES	57.8	15.7	16.5
FLIGHTS	61.6	8.76	9.89
WILDFIRES	0.28	0.12	0.14
BOOKS	1.86	0.85	0.87

$[q.tstart, q.tend]$ and then, checks the attribute A of every intermediate result against the $[q.Astart, q.Aend]$ range. The second competitor is the state-of-the-art index for multi-versioned DBs, MVB-tree [37]. The first and the third rows in Figure 6.10 report the total time of the indices, while varying the A-range of the query and the temporal query extent, respectively. Observe that both LIT-based indices outperform the MVB-tree, in all input streams and tests. The reason is the high cost of update handling by the MVB-tree; the performance gap is larger for the TAXIS and BIKES (update-heavy streams). As Table 6.9 shows, LIT (pure) and a-LIT capitalize on the LiveIndex to cope with updates. In fact the MVB-tree is competitive only in BOOKS; this dataset has the smallest number of updates and queries significantly contribute to the total time. a-LIT always outperforms LIT (pure) as expected in answering range time-travel queries (second and fourth row in Figure 6.10), because LIT (pure) cannot prune the search space using the search-key attribute. Overall, a-LIT exhibits a good tradeoff between updating and querying, being able to efficiently handle both update-heavy and moderate streams. Based on the our tests, we expect a even bigger advantage over LIT (pure) for query-heavy streams.

6.5.4 Index Size

We conclude our analysis with the index size growth over time. Figure 13 plots LIT's size as a function of the percentage of the updates in each stream. Observe that LIT's space increases linearly with the number of updates, which makes it appropriate for in-memory management of time-evolving data. A linear growth ensures that its performance characteristics remain consistent. As the number of records increases, the time complexity of index lookups remains relatively constant, allowing for consistent and reliable query performance. Furthermore, as new records are added to the database, the index can be updated in a predictable manner, making it easier to maintain. Finally, the index can handle an increasing amount of data without a significant drop in query performance, which makes it scalable.

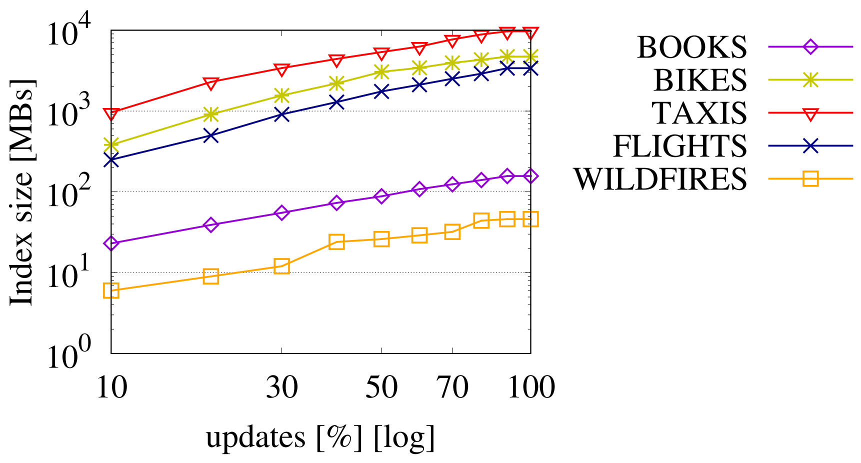


Figure 6.11: LIT: size growth over time

6.6 Conclusions and Future Work

In this chapter, we proposed LIT, a hybrid index for time-evolving databases, which decouples the handling of current (live) record versions from the management of past (dead) record versions. We studied options for implementing the live and dead index components, focusing on minimizing the cost of index updates and queries. We considered pure time-travel queries that retrieve active record versions at some time point or period in the past, and range time-travel queries, which additionally apply a selection predicate on a search-key attribute. Our tests unveil the best approaches for handling live and dead record versions in LIT and shows that LIT is orders of magnitude faster than temporal indices that index live and dead versions in the same structure. LIT uses linear space to the number of record versions, which renders it suitable for in-memory indexing of temporal data.

Chapter 7

Conclusions and future work

In conclusion, we present a summary of our significant contributions, along with outlining potential paths for future research.

7.1 Summary of Contributions

In this dissertation we studied interval data management in main memory. Interval data find many applications. Yet, as we showed the multiple use cases and query operators were not sufficiently covered by the previous work in the area. Since indexing plays a crucial role in performance, database systems have high demand for fast in-memory implementations of indices with widely-used query operators.

In the first part of our work we analyzed the use case of indexing intervals with valid time specifications. We proposed a hierarchical index (HINT) for intervals, which has low space complexity and minimizes the number of data accesses and comparisons during query evaluation. We introduced a division of intervals into groups depending on how their endpoints match the partition boundaries, we theoretically proved that the expected partitions for which comparisons are necessary is at most four, we added further optimizations for fast retrieval times and we proposed a model for tuning the value of the parameter m for HINT^m. Our experimental evaluation on real and synthetic datasets showed that HINT outperforms previous work by almost one order of magnitude in a wide variety of interval data and query distributions. Furthermore, we evaluated our methods for interval join. We conducted experiments against the state-of-the-art and showed that for small sized datasets, HINT^m is able to outperform the competition.

In the second part of our work, we extented our index so that it can answer effieciently Allen's predicates. We showed the necessary additional comparisons and accesses on HINT^m for each predicate in Allen's algebra. In addition, we showed that a generalized version of HINT^m is directly suitable for processing queries using all Allen's predicates, while maintaining the excellent performance of HINT^m for G-OVERLAPS queries. Our index fully supports selection queries based on Allen's relationships [72] between intervals, achieving consistently excellent performance independently of the query predicate.

In the third part of this dissertation we studied indexing for transaction-time databases. We proposed LIT, a hybrid index for time-evolving data, which decouples the handling of current (live) record versions from the management of past (dead) record versions. We studied options for implementing the live and dead index components, focusing on minimizing the cost of index updates and queries. We considered pure time-travel queries that retrieve active record versions at some time point or period in the past, and range time-travel queries, which additionally apply a selection predicate on a search-key attribute. Our experimental evaluation unveiled the best approaches for handling live and dead record versions in LIT and shows that LIT is orders of magnitude faster than temporal indices that index live and dead versions in the same structure. We also showed that LIT uses linear space to the number of record versions, which renders it suitable for in-memory indexing of temporal data.

7.2 Directions for Future Work

In this section, we outline ideas for additional research. For future work, there are several directions, on which we elaborate below:

Multiple Temporal Operators. Most proposed indices are specialized on one temporal operator. Keeping a different index for each type of query is not affordable for a

DBMS in terms of tuning, maintenance and storage overhead. Different indices will have different beneficial sortings, will possibly cause data replication and different optimizations in general. All the commonly used operators, temporal aggregation, time travel and temporal join should be supported by one index. Eventually, we plan to generalize LIT to support all the commonly used operators so that it becomes a versatile index capable of temporal database system integration. This direction will have an impact on the information kept for the intervals and may lead to the use of additional auxiliary indices for supporting the new operators.

Distributed Computation. Moreover, the big volumes of data may be distributed among multiple physical locations. This makes exploring the integration of distributed temporal indexing within the Apache Spark [74] an interesting research direction for future work. The key-challenge is to align our algorithms of data partitioning with Spark practices. First of all, we plan to prepare known collections of intervals with labels of partitions (and groups) they would be assigned to if we would utilize a HINT structure. Then the data will be organized with functions available in Spark (e.q. groupBy) and grouped in Dataframes. This will ensure that our indexing will be aligned with Spark's data partitioning scheme. By utilizing Spark's transformations, we can efficiently organize the indexed data into different computing nodes. Next steps, contain optimizing query performance and maintaining the index in an incremental manner, using Spark's built-in functions and leveraging Spark Streaming.

Temporal Database Management System (DBMS) Design. Another direction for future work is the integration of our algorithms in a DBMS like PostgreSQL. We plan to add the algorithms and partitioning scheme of HINT in PostgreSQL by extending the Generalized Search Tree. GiST is purely compatible with the query planner, optimizer, and execution engine of PostgreSQL, while being flexible and versatile. By extending the capabilities of the GiST index with specialized GiST operator classes, tailored extractors, and comparators designed for interval data, more efficient handling of range interval queries can be achieved. Additionally, with the adoption of PostgreSQL's built-in interval partitioning mechanisms we plan to enable the logical organization of interval data into partitions, further optimizing data retrieval for range interval queries.

Bibliography

[1] L. H. U, N. Mamoulis, K. Berberich, and S. J. Bedathur, "Durable top-k search in document archives," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6-10, 2010*. ACM, 2010, pp. 555–566.

[2] W. Lu, Z. Zhao, X. Wang, H. Li, Z. Zhang, Z. Shui, S. Ye, A. Pan, and X. Du, "A lightweight and efficient temporal database management system in TDSQL," *Proc. VLDB Endow.*, vol. 12, no. 12, pp. 2035–2046, 2019.

[3] N. N. Dalvi and D. Suciu, "Efficient query evaluation on probabilistic databases," in *VLDB*, 2004, pp. 864–875.

[4] P. Samarati and L. Sweeney, "Generalizing data to provide anonymity when disclosing information (abstract)," in *ACM PODS*, 1998, p. 188.

[5] J. Min, M. Park, and C. Chung, "XPRESS: A queriable compression for XML data," in *ACM SIGMOD*, 2003, pp. 122–133.

[6] J. F. Allen, "An interval-based representation of temporal knowledge," in *IJCAI*, 1981, pp. 221–226.

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

[8] A. Awad, R. Tommasini, S. Langhi, M. Kamel, E. D. Valle, and S. Sakr, "D^2IA: user-defined interval analytics on distributed streams," *Information Systems*, vol. 104, p. 101679, 2022.

[9] R. T. Snodgrass and I. Ahn, "Temporal databases," *Computer*, vol. 19, no. 9, pp. 35–42, 1986.

[10] M. H. Böhlen, A. Dignös, J. Gamper, and C. S. Jensen, "Temporal data management - an overview," in *eBISS*, 2017, pp. 51–83.

[11] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo, "Coalescing in temporal databases," in *VLDB*, 1996, pp. 180–191.

[12] D. Gao, C. S. Jensen, R. T. Snodgrass, and M. D. Soo, "Join operations in temporal databases," *VLDB J.*, vol. 14, no. 1, pp. 2–29, 2005.

[13] M. Kaufmann, A. A. Manjili, P. Vagenas, P. M. Fischer, D. Kossmann, F. Färber, and N. May, "Timeline index: a unified data structure for processing queries on temporal data in SAP HANA," in *ACM SIGMOD*, 2013, pp. 1173–1184.

[14] A. Dignös, B. Glavic, X. Niu, J. Gamper, and M. H. Böhlen, "Snapshot semantics for temporal multiset relations," *Proc. VLDB Endow.*, vol. 12, no. 6, pp. 639–652, 2019. [Online]. Available: http://www.vldb.org/pvldb/vol12/p639-dignoes.pdf

[15] K. Papaioannou, M. Theobald, and M. H. Böhlen, "Outer and anti joins in temporal-probabilistic databases," in *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019.* IEEE, 2019, pp. 1742–1745. [Online]. Available: https://doi.org/10.1109/ICDE.2019.00187

[16] J. Gao, S. Sintos, P. K. Agarwal, and J. Yang, "Durable top-k instant-stamped temporal records with user-specified scoring functions," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 2021, pp. 720–731. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00068

[17] Z. Zhang, H. Hu, Z. Xue, C. Chen, Y. Yu, C. Fu, X. Zhou, and F. Li, "SLIMSTORE: A cloud-based deduplication system for multi-version backups," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 2021, pp. 1841–1846. [Online]. Available: https://doi.org/10.1109/ICDE51399.2021.00164

[18] L. Bellomarini, M. Nissl, and E. Sallinger, "itemporal: An extensible generator of temporal benchmarks," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022.* IEEE, 2022, pp. 2021–2033. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00197

[19] A. Bernhardt, S. Tamimi, T. Vinçon, C. Knödler, F. Stock, C. Heinz, A. Koch, and I. Petrov, "neodbms: In-situ snapshots for multi-version DBMS on native computational storage," in *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 2022, pp. 3170–3173. [Online]. Available: https://doi.org/10.1109/ICDE53745.2022.00290

[20] F. S. Campbell, B. S. Arab, and B. Glavic, "Efficient answering of historical what-if queries," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 1556–1569. [Online]. Available: https://doi.org/10.1145/3514221.3526138

[21] X. Hu, S. Sintos, J. Gao, P. K. Agarwal, and J. Yang, "Computing complex temporal join queries efficiently," in *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Z. G. Ives, A. Bonifati, and A. E. Abbadi, Eds. ACM, 2022, pp. 2076–2090. [Online]. Available: https://doi.org/10.1145/3514221.3517893

[22] L. Bornemann, T. Bleifuß, D. V. Kalashnikov, F. Nargesian, F. Naumann, and D. Srivastava, "Matching roles from temporal data: Why joe biden is not only president, but also commander-in-chief," *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 65:1–65:26, 2023. [Online]. Available: https://doi.org/10.1145/3588919

[23] C. S. Jensen and R. T. Snodgrass, "Temporal data management," *IEEE Trans. Knowl. Data Eng.*, vol. 11, no. 1, pp. 36–44, 1999.

[24] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Comput. Surv.*, vol. 31, no. 2, pp. 158–221, 1999.

[25] A. Dignös, M. H. Böhlen, and J. Gamper, "Overlap interval partition join," in *ACM SIGMOD*, 2014, pp. 1459–1470.

[26] D. Piatov, S. Helmer, and A. Dignös, "An interval join optimized for modern hardware," in *IEEE ICDE*, 2016, pp. 1098–1109.

[27] P. Bouros and N. Mamoulis, "A forward scan based plane sweep algorithm for parallel interval joins," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1346–1357, 2017.

[28] P. Bouros, N. Mamoulis, D. Tsitsigkos, and M. Terrovitis, "In-memory interval joins," *VLDB J.*, vol. 30, no. 4, 2021.

[29] F. Cafagna and M. H. Böhlen, "Disjoint interval partitioning," *VLDB J.*, vol. 26, no. 3, pp. 447–466, 2017.

[30] N. Kline and R. T. Snodgrass, "Computing temporal aggregates," in *IEEE ICDE*, 1995, pp. 222–231.

[31] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger, "Efficient computation of temporal aggregates with range predicates," in *ACM PODS*, 2001.

[32] B. Moon, I. F. V. López, and V. Immanuel, "Efficient algorithms for large-scale temporal aggregation," *IEEE TKDE*, vol. 15, no. 3, pp. 744–759, 2003.

[33] Y. Tao, D. Papadias, and C. Faloutsos, "Approximate temporal aggregation," in *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004, 30 March - 2 April 2004, Boston, MA, USA*, Z. M. Özsoyoglu and S. B. Zdonik, Eds., 2004, pp. 190–201.

[34] D. Piatov and S. Helmer, "Sweeping-based temporal aggregation," in *SSTD*, 2017, pp. 125–144.

[35] H. Edelsbrunner, "Dynamic rectangle intersection searching," Institute for Information Processing, Technical University of Graz, Austria, Tech. Rep. 47, 1980.

[36] A. Behrend, A. Dignös, J. Gamper, P. Schmiegelt, H. Voigt, M. Rottmann, and K. Kahl, "Period index: A learned 2d hash index for range and duration queries," in *SSTD*, 2019, pp. 100–109.

[37] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer, "An asymptotically optimal multiversion b-tree," *VLDB J.*, vol. 5, no. 4, pp. 264–275, 1996.

[38] M. M. Moro and V. J. Tsotras, "Valid-time indexing," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[39] H. Kriegel, M. Pötke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB*, 2000, pp. 407–418.

[40] G. Christodoulou, P. Bouros, and N. Mamoulis, "HINT: A hierarchical index for intervals in main memory," in *ACM SIGMOD*, 2022, pp. 1257–1270.

[41] J. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *IEEE ICDE*, 2000, pp. 535–546.

[42] D. Tsitsigkos, K. Lampropoulos, P. Bouros, N. Mamoulis, and M. Terrovitis, "A two-layer partitioning for non-point spatial data," in *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 2021, pp. 1787–1798.

[43] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984.* ACM Press, 1984, pp. 47–57.

[44] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25, 1990.* ACM Press, 1990, pp. 322–331.

[45] V. Gaede and O. Günther, "Multidimensional access methods," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 170–231, 1998.

[46] M. W. Chekol, G. Pirrò, and H. Stuckenschmidt, "Fast interval joins for temporal SPARQL queries," in *ACM WWW*, 2019, pp. 1148–1154.

[47] K. Zhu, G. H. L. Fletcher, N. Yakovets, O. Papapetrou, and Y. Wu, "Scalable temporal clique enumeration," in *SSTD*, 2019, pp. 120–129.

[48] P. Bouros, K. Lampropoulos, D. Tsitsigkos, N. Mamoulis, and M. Terrovitis, "Band joins for interval data," in *EDBT*, 2020, pp. 443–446.

[49] P. Bouros and N. Mamoulis, "Interval count semi-joins," in *EDBT*, 2018, pp. 425–428.

[50] M. M. Moro and V. J. Tsotras, "Transaction-time indexing," in *Encyclopedia of Database Systems, Second Edition*, L. Liu and M. T. Özsu, Eds. Springer, 2018.

[51] 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, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, D. McLeod, R. Sacks-Davis, and H. Schek, Eds. Morgan Kaufmann, 1990, pp. 1–12. [Online]. Available: http://www.vldb.org/conf/1990/P001.PDF

[52] D. B. Lomet and B. Salzberg, "Access methods for multiversion data," in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA.* ACM Press, 1989, pp. 315–324.

[53] ——, "The performance of a multiversion access method," in *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, USA*, 1990, pp. 353–363.

[54] M. Stonebraker, "The design of the POSTGRES storage system," in *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, 1987, pp. 289–300.

[55] S. Lanka and E. Mays, "Fully persistent b+-trees," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, USA*, 1991, pp. 426–435.

[56] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *J. Comput. Syst. Sci.*, vol. 38, no. 1, pp. 86–124, 1989.

[57] D. Piatov, S. Helmer, A. Dignös, and F. Persia, "Cache-efficient sweeping-based interval joins for extended allen relation predicates," *VLDB J.*, vol. 30, no. 3, pp. 379–402, 2021.

[58] C. Gutierrez, C. A. Hurtado, and A. A. Vaisman, "Introducing time into RDF," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 2, pp. 207–218, 2007. [Online]. Available: https://doi.org/10.1109/TKDE.2007.34

[59] K. Bereta, P. Smeros, and M. Koubarakis, "Representation and querying of valid time of triples in linked geospatial data," in *The Semantic Web: Semantics and Big Data, 10th International Conference, ESWC 2013, Montpellier, France, May 26-30, 2013. Proceedings*, ser. Lecture Notes in Computer Science, P. Cimiano, Ó. Corcho, V. Presutti, L. Hollink, and S. Rudolph, Eds., vol. 7882. Springer, 2013, pp. 259–274. [Online]. Available: https://doi.org/10.1007/978-3-642-38288-8_18

[60] J. Mylopoulos, A. Borgida, M. Jarke, and M. Koubarakis, "Telos: Representing knowledge about information systems," *ACM Trans. Inf. Syst.*, vol. 8, no. 4, pp. 325–362, 1990. [Online]. Available: https://doi.org/10.1145/102675.102676

[61] C. Saracco, M. Nicola, and L. Gandhi. (2012) A matter of time: Temporal data management in db2 10. http://www.ibm.com/developerworks/data/library/techarticle/dm-1204db2temporaldata/dm-1204db2temporaldata-pdf.pdf.

[62] Teradata. (2014) Teradata database 14.10 - temporal table support. http://www.info.teradata.com/eDownload.cfm?itemid=131540028.

[63] Oracle. (2016) Database development guide - temporal validity support. https://docs.oracle.com/database/121/ADFNS/adfns_design.htm#ADFNS967.

[64] J. Davis. (2009) Online temporal postgresql reference. http://temporal.projects.postgresql.org/reference.html.

[65] PostgreSQL Global Development Group. (2012) Documentation manual: Postgresql - range types. http://www.postgresql.org/docs/9.2/static/rangetypes.html.

[66] B. Pagel, H. Six, H. Toben, and P. Widmayer, "Towards an analysis of range query performance in spatial data structures," in *ACM PODS*, 1993, pp. 214–221.

[67] D. B. Lomet, "Scheme for invalidating references to freed storage," *IBM J. Res. Dev.*, vol. 19, no. 1, pp. 26–35, 1975.

[68] M. H. Overmars, *The Design of Dynamic Data Structures*, ser. Lecture Notes in Computer Science. Springer, 1983, vol. 156.

[69] P. Ferragina and G. Vinciguerra, "The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endow.*, vol. 13, no. 8, pp. 1162–1175, 2020.

[70] E. Garrison, "A minimal c++ interval tree implementation," https://github.com/ekg/intervaltree.

[71] A. Monacchi, D. Egarter, W. Elmenreich, S. D'Alessandro, and A. M. Tonello, "GREEND: an energy consumption dataset of households in italy and austria," in *SmartGridComm*, 2014, pp. 511–516.

[72] J. F. Allen, "Maintaining knowledge about temporal intervals," *Commun. ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[73] M. Loskot and A. Wulkiewicz, 2019, https://github.com/mloskot/spatial_index_benchmark.

[74] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster computing with working sets,” in *2nd USENIX Workshop on Hot Topics in Cloud Computing, HotCloud’10, Boston, MA, USA, June 22, 2010*. USENIX Association, 2010.

Author's Publications

- George Christodoulou, Panagiotis Bouros, Nikos Mamoulis, **LIT: Lightning-fast In-memory Temporal Indexing**, submitted to *SIGMOD 2024*, Santiago, Chile
- George Christodoulou, **Efficient And Scalable Management Of Interval Data** in *PhD Workshop* in *EDBT'23*, Ioannina, Greece
- George Christodoulou, Panagiotis Bouros, Nikos Mamoulis, **HINT: A Hierarchical Interval Index for Allen Relationships** in *VLDBJ' 23*
- George Christodoulou, Panagiotis Bouros, Nikos Mamoulis, **HINT: A Hierarchical Index for Intervals in Main Memory** in *SIGMOD'22*, Philadelphia, USA

Short Biography

George Christodoulou was born in 1993. He received his diploma in Computer Science and Engineering from the Computer Science and Engineering Department of the University of Ioannina in 2017. He received his MSc in Computer Science with specialization in software from the same department, supervised by Prof Mamoulis. During his PhD studies, he went to Mainz (Germany) as an intern at the Institute of Computer Science of the University of Mainz working with Prof. Bouros. Furthermore he went to Hong Kong as an intern at the Hong Kong University of Science and Technology (HKUST)working with Prof. Papadias. His research interests include Data Management, Query processing (Efficient Data Indexing and Retrieval on temporal data) and Database Management System engines.