# In-Memory Indexing for Parallel Processing of Single and Multi-Dimensional Queries

A Dissertation

submitted to the designated

by the Assembly

of the Department of Computer Science and Engineering

Examination Committee

by

Dimitris Tsitsigkos

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

University of Ioannina

School of Engineering

Ioannina 2024

Advisory Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (advisor)
- **Panagiotis Vassiliadis**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Manolis Terrovitis**, Researcher, Information Management Systems Institute (IMSI) of the Research and Innovation Centre in Information, Communication and Knowledge Technologies ”Athena”

Examining Committee:

- **Nikos Mamoulis**, Professor, Department of Computer Science and Engineering, University of Ioannina (advisor)
- **Panagiotis Vassiliadis**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Manolis Terrovitis**, Researcher, Information Management Systems Institute (IMSI) of the Research and Innovation Centre in Information, Communication and Knowledge Technologies ”Athena”
- **Evaggelia Pitoura**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Vassilios V. Dimakopoulos**, Professor, Department of Computer Science and Engineering, University of Ioannina
- **Panagiotis Bouros**, Assistant Professor, Institute of Computer Science, Johannes Gutenberg University Mainz
- **Christos Doulkeridis**, Professor, Department of Digital Systems, University of Piraeus

# Dedication

*To my beloved family*

# Acknowledgements

First, I would like to thank my advisor, Nikos Mamoulis, for his guidance, support, and for giving me this opportunity. His patience and encouragement were invaluable throughout this challenging journey. I would also like to sincerely thank Panagiotis Bouros for his help and for sharing his technical expertise, which greatly helped me improve my dissertation. Additionally, I am very grateful to Manolis Terrovitis for believing in me from the start and supporting me throughout this process.

I also want to thank my colleagues (Chrysanthi Kosyfaki, Dinos Lampropoulos, George Christodoulou, Thanasis Georgiadis, Achilleas Michalopoulos) in the lab for their interesting discussions, both scientific and casual, and for creating such a positive and inspiring environment. A special thanks to Achilleas for our collaborations, which were both productive and valuable.

I am deeply thankful to Xenia for her constant support and patience during my studies. She has always been there for me, encouraging me through both the good and the tough times.

I would also like to thank my friends for their understanding and patience, especially during the long periods when my busy schedule made it hard to stay in touch.

Finally, I want to express my deepest gratitude to my family, especially my mother, Eleni, whose strong belief in me has always been a source of strength and inspiration. Her support and encouragement have helped me overcome every challenge, and I will be forever grateful.

*Dimitris Tsitsigkos*

*December 2024, Ioannina*

# Table of Contents

# List of Figures

# List of Tables

# List of Algorithms

# Abstract

Dimitris Tsitsigkos, Ph.D., Department of Computer Science and Engineering, School of Engineering, University of Ioannina, Greece, 2024.
In-Memory Indexing for Parallel Processing of Single and Multi-Dimensional Queries.
Advisor: Nikos Mamoulis, Professor.

Database Systems are essential for modern applications, providing structured, efficient, and reliable ways to manage data in fields like banking, healthcare, and scientific research. They are designed to handle large amounts of data and answer queries quickly. A key factor in making queries fast is the use of indices. Efficient indexing can significantly improve the speed, credibility, and overall performance of a database. A good index supports fast searches and updates, while having low space requirements. With the advancements in hardware, we can redesign index structures to be faster and more efficient. For example, larger memory capacities allow us to move computations from disk storage to faster in-memory processing. Additionally, the latest processors offer significantly more cores, enhancing parallel computing by distributing tasks across multiple cores to improve performance. These modern hardware components are affordable and found in commodity computers, making them accessible for a wide range of applications.

This dissertation examines how in-memory indexing combined with parallel processing techniques can enhance the performance of relational, temporal, and spatial databases. It addresses challenges like managing large-scale data and running complex queries on modern systems.

Interval joins are crucial for temporal databases and are also useful in many applications. However, a major challenge in parallel implementations of interval joins is dividing the workload effectively across processor cores. A simple approach is to split the data domain into disjoint partitions and assign the data in each partition to a different core. However, this creates a new problem: when an interval spans

multiple stripes, it must be processed by multiple cores, possibly generating duplicate join results. One way to handle duplicates is by using a data structure like a set to store unique results. However, this increases memory usage and slows down query performance because of the extra checks for duplicates. Domain-based partitioning approach divides the data into partitions that can be processed independently and in parallel, maximizing multi-core hardware capabilities. This study introduces three distinct strategies for parallel partitioning applicable to both hash-based and domain-based methods, significantly accelerating partitioning operations by reducing computational overhead and improving scalability.

Indexing spatial data is challenging due to their shape and dimensionality. This complexity affects query performance, particularly for spatial intersection joins, which are the most resource-intensive. Like temporal databases, spatial databases also face duplication issues. For example, in a 2D grid, duplicates can occur when spatial objects overlap multiple grid tiles. The most common technique to avoid duplicates is the reference point approach, which reports a result in only one tile of the grid but requires additional computations. This dissertation primarily focuses on non-point data, where duplication issues arise. Our first study focuses on improving the Partition-Based Spatial Join (PBSM) algorithm for in-memory and parallel evaluation of spatial joins. We show how to choose the best partitioning settings based on data statistics to fine-tune the algorithm for specific join inputs. In our second study, we introduce a new secondary partitioning technique for space-oriented partitioning (SOP) indices, such as grids. This technique removes duplicate results during spatial queries by organizing objects within each partition and only accessing classes that do not produce duplicates. This innovation greatly improves the efficiency of grid-based spatial indices for range (disk and rectangle) and intersection joins. Finally, we propose a parallel method that boosts the performance of range queries.

Relational databases also face challenges in creating efficient indices. This dissertation focuses only on tree-like indices. One of the most well-known and efficient indexing structures in this domain is the $B^+$-tree, particularly suited for skewed workloads with dynamic data and for supporting range queries. With the rise of machine learning, many learned indices have been introduced. A learned index uses ML models to "learn" the data distribution and predict the location of the search key within a dataset aiming to reduce the space requirements and the memory accesses. The main difference between a traditional $B^+$-tree and a learned index is that learned

indices replace inner nodes with machine learning models. We believe that B$^+$-tree are more efficient than learned indices because their query performance is stable and not affected by the data distribution. We also believe that advancements in hardware technology create new opportunities to further improve B-tree performance. Building on the foundational B$^+$-tree structure, this research introduces B$^S$-tree, a new indexing structure designed for main memory and modern hardware. The B$^S$-tree leverages data parallelism and integrates innovative optimizations, offering significant advancements over both traditional B$^+$-tree and emerging learned indices. Key features include a data-parallel branching mechanism implemented using SIMD instructions, a gap management strategy employing duplicate keys to delay splits and reduce data-shifting overhead, and a node compression scheme that minimizes memory usage while maintaining high throughput.

In summary, this dissertation provides a comprehensive framework for enhancing the performance of relational, temporal, and spatial databases. Through the integration of in-memory indexing, parallel processing techniques and duplicate avoidance techniques, this dissertation delivers robust, scalable, and efficient solutions for the evolving needs of modern data-intensive applications.

# Εκτεταμενη Περιληψη

Δημήτρης Τσιτσίγκος, Δ.Δ., Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πολυτεχνική Σχολή, Πανεπιστήμιο Ιωαννίνων, 2024.
Ευρετηρίαση στη Μνήμη για Παράλληλη Επεξεργασία Ερωτημάτων Μονοδιάστατων και Πολυδιάστατων Δεδομένων.
Επιβλέπων: Νίκος Μαμουλής, Καθηγητής.

Οι βάσεις δεδομένων είναι απαραίτητες για τις σύγχρονες εφαρμογές, προσφέροντας δομημένους, αποδοτικούς και αξιόπιστους τρόπους διαχείρισης δεδομένων σε τομείς όπως οι τραπεζικές συναλλαγές, η υγειονομική περίθαλψη και η επιστημονική έρευνα. Έχουν σχεδιαστεί για να διαχειρίζονται μεγάλους όγκους δεδομένων και να απαντούν αποδοτικά σε ερωτήματα. Βασικός παράγοντας που επηρεάζει την ταχύτητα των ερωτημάτων είναι η χρήση ευρετηρίων, τα οποία μπορούν να βελτιώσουν σημαντικά την ταχύτητα, την αξιοπιστία και τη συνολική απόδοση μιας βάσης δεδομένων. Ένα αποδοτικό ευρετήριο θα πρέπει να υποστηρίζει γρήγορες αναζητήσεις και ενημερώσεις, ενώ ταυτόχρονα θα πρέπει να διατηρεί χαμηλές απαιτήσεις σε αποθηκευτικό χώρο. Με τις τεχνολογικές εξελίξεις στο υλικό (hardware) των υπολογιστών, μπορούμε να ανασχεδιάσουμε τις δομές των ευρετηρίων, ώστε να γίνουν ταχύτερα και πιο αποδοτικά. Για παράδειγμα, η αυξημένη χωρητικότητα της κύριας μνήμης επιτρέπει τη μεταφορά υπολογισμών από τον δίσκο, ένα αργό αποθηκευτικό μέσο, στη μνήμη, όπου οι υπολογισμοί μπορούν να εκτελούνται ταχύτερα. Παράλληλα, οι σύγχρονοι επεξεργαστές διαθέτουν περισσότερους πυρήνες, ενισχύοντας την παράλληλη εκτέλεση των εργασιών σε πολλαπλούς πυρήνες, βελτιώνοντας έτσι την απόδοση. Επιπλέον, αυτές οι σύγχρονες τεχνολογίες υλικού είναι πλέον οικονομικά προσιτές και διαθέσιμες ακόμα και σε συμβατικούς υπολογιστές, καθιστώντας τες ιδανικούς για πληθώρα εφαρμογών.

Αυτή η διατριβή εξετάζει πώς οι τεχνικές παράλληλης δεικτοδότησης στη κύρια μνήμη μπορούν να βελτιώσουν την απόδοση των σχεσιακών, χρονικών και χωρικών

βάσεων δεδομένων. Αντιμετωπίζει προκλήσεις όπως η διαχείριση δεδομένων μεγάλης κλίμακας και η εκτέλεση πολύπλοκων ερωτημάτων σε σύγχρονα συστήματα.

Οι συνενώσεις διαστημάτων (interval joins) αποτελούν σημαντικό κομμάτι των χρονικών βάσεων δεδομένων και είναι χρήσιμες σε πολλές εφαρμογές. Ωστόσο, μια σημαντική πρόκληση στις παράλληλες υλοποιήσεις της συνένωσης διαστημάτων είναι η αποτελεσματική κατάτμηση του χώρου, με στόχο την καλύτερη κατανομή του φόρτου εργασίας στους πυρήνες του επεξεργαστή. Μια απλή προσέγγιση είναι η διαίρεση του χώρου δεδομένων σε μη επικαλυπτόμενες περιοχές και η ανάθεση των δεδομένων κάθε περιοχής σε διαφορετικό πυρήνα. Ωστόσο, αυτό δημιουργεί ένα νέο πρόβλημα: όταν ένα διάστημα εκτείνεται σε πολλές περιοχές, η επεξεργασία των διαστημάτων αυτών από διαφορετικούς πυρήνες, μπορεί να οδηγήσει σε διπλότυπα αποτελέσματα. Ένας τρόπος διαχείρισης των διπλοτύπων είναι η χρήση μιας δομής δεδομένων, όπως ένα σύνολο (set), για την αποθήκευση μοναδικών αποτελεσμάτων. Ωστόσο, αυτό αυξάνει τη χρήση μνήμης και επιβραδύνει την απόδοση των ερωτημάτων λόγω των επιπλέον ελέγχων για διπλότυπα. Μια αποδοτική τεχνική για τον διαχωρισμό των δεδομένων, που επιλύει τα παραπάνω προβλήματα, είναι η προσέγγιση κατανομής του χώρου δεδομένων (Domain-based partitioning). Αυτή η μέθοδος διαιρεί τα δεδομένα σε τμήματα που μπορούν να υποστούν επεξεργασία ανεξάρτητα και παράλληλα, αξιοποιώντας στο έπακρο τις δυνατότητες των πολυπύρηνων επεξεργαστών. Σε αυτή την διατριβή προτείνονται τρεις παράλληλες στρατηγικές, οι οποίες είναι εφαρμόσιμες τόσο σε μεθόδους βασισμένες σε κατακερματισμό (hash-based) όσο και σε κατανομές του χώρου δεδομένων (domain-based). Αυτές οι στρατηγικές επιταχύνουν σημαντικά τις λειτουργίες συνένωσης διαστημάτων, ελαχιστοποιώντας το υπολογιστικό κόστος και βελτιώνοντας την κλιμακωσιμότητα.

Η ευρετηρίαση χωρικών δεδομένων είναι απαιτητική λόγω της πολυπλοκότητας του σχήματος και της πολυδιάστατης φύσης τους. Αυτή η πολυπλοκότητα επηρεάζει την απόδοση των ερωτημάτων, ιδιαίτερα των χωρικών συνενώσεων (spatial intersection joins), οι οποίες απαιτούν πολλούς υπολογιστικούς πόρους. Όπως και οι χρονικές βάσεις δεδομένων, οι χωρικές βάσεις δεδομένων αντιμετωπίζουν επίσης προβλήματα διπλότυπων. Για παράδειγμα, σε ένα δισδιάστατο πλέγμα (grid), διπλότυπα μπορεί να εμφανιστούν όταν χωρικά αντικείμενα επικαλύπτουν πολλαπλές διαμερίσεις του πλέγματος. Η πιο κοινή τεχνική για την αποφυγή διπλοτύπων είναι η προσέγγιση σημείου αναφοράς (reference point approach), η οποία αναφέρει ένα

αποτέλεσμα μόνο σε μία διαμέριση του πλέγματος, αλλά απαιτεί επιπλέον υπολογισμούς. Αυτή η διατριβή επικεντρώνεται κυρίως σε μη σημειακά δεδομένα, όπου προκύπτουν ζητήματα διπλοτύπων αποτελεσμάτων. Η πρώτη μας μελέτη εστιάζει στη βελτίωση του αλγορίθμου Partition-Based Spatial Join (PBSM) για την αξιολόγηση ερωτημάτων συνένωσης στην κύρια μνήμη με παραλληλοποίηση. Επίσης, προτείνουμε τρόπους για την επιλογή των παραμέτρων του αλγορίθμου, χρησιμοποιώντας στατιστικά δεδομένα του συνόλου δεδομένων. Στη δεύτερη μελέτη μας, εισάγουμε μια νέα τεχνική κατανομής για ευρετήρια κατανομής χώρου (space-oriented partitioning - SOP), όπως τα πλέγματα. Αυτή η τεχνική δημιουργεί μια καινούργια κατανομή πάνω από το πλέγμα και κατηγοριοποιεί τα αντικείμενα σε κλάσεις. Η αποφυγή των διπλοτύπων αποτελεσμάτων γίνεται κατά την εκτέλεση του χωρικού ερωτήματος, όπου ο αλγόριθμος μας προσπελαύνει μόνο τις κλάσεις που δεν παράγουν διπλότυπα. Αυτή η καινοτομία βελτιώνει σημαντικά την αποδοτικότητα των ευρετηρίων πλέγματος για ερωτήματα εύρους και συνενώσεων. Τέλος, προτείνουμε μια παράλληλη μέθοδο που ενισχύει την απόδοση των ερωτημάτων εύρους.

Οι σχεσιακές βάσεις δεδομένων αντιμετωπίζουν επίσης προκλήσεις στη δημιουργία αποδοτικών ευρετηρίων. Αυτή η διατριβή επικεντρώνεται αποκλειστικά σε δενδρικές δομές ευρετηρίων. Μία από τις πιο γνωστές και αποδοτικές δομές ευρετηρίου σε αυτόν τον τομέα είναι το $B^+$-δέντρο, το οποίο είναι ιδιαίτερα κατάλληλο για άνισα φορτία εργασίας με δυναμικά δεδομένα και για την υποστήριξη ερωτημάτων εύρους. Με την εξέλιξη της μηχανικής μάθησης (machine learning), έχουν προταθεί πολλά ευρετήρια (learned indices) που χρησιμοποιούν μοντέλα μηχανικής μάθησης για να κατανοήσουν την κατανομή των δεδομένων και να προβλέψουν τη θέση του κλειδιού αναζήτησης μέσα σε ένα σύνολο δεδομένων. Τα ευρετήρια αυτά στοχεύουν στη μείωση των απαιτήσεων χώρου και των προσβάσεων στη μνήμη. Η κύρια διαφορά μεταξύ ενός παραδοσιακού $B^+$-δέντρου και ενός learned index είναι ότι τα learned indices αντικαθιστούν τους εσωτερικούς κόμβους με μοντέλα μηχανικής μάθησης. Πιστεύουμε ότι τα $B^+$-δέντρα είναι πιο αποδοτικά από τα learned indices, επειδή η απόδοση των ερωτημάτων τους είναι σταθερή και δεν επηρεάζεται από την κατανομή των δεδομένων. Πιστεύουμε επίσης ότι οι σύγχρονες τεχνολογίες υλικού δημιουργούν νέες ευκαιρίες για περαιτέρω βελτίωση της απόδοσης του $B^+$-δέντρου. Βασισμένη στη βασική δομή του $B^+$-δέντρο, η έρευνά μας εισάγει το $B^S$-δέντρο, ένα νέο ευρετήριο σχεδιασμένο για την κύρια μνήμη, το οποίο αξιοποιεί σύγχρονες τεχνολογίες. Το $B^S$-δέντρο εκμεταλλεύεται την παραλληλία δεδομένων και εν-

σωματώνει καινοτόμες βελτιστοποιήσεις, προσφέροντας σημαντικές βελτιώσεις σε σχέση τόσο με τα παραδοσιακά $B^+$-δέντρα όσο και με learned indices. Τα κύρια χαρακτηριστικά του, περιλαμβάνουν έναν παράλληλο μηχανισμό διακλάδωσης με χρήση εντολών SIMD, μια στρατηγική διαχείρισης άδειων θέσεων που χρησιμοποιεί διπλότυπα κλειδιά για να καθυστερήσει τον διαχωρισμό κόμβων και να μειώσει το κόστος αναδιάταξης δεδομένων. Τέλος, προτείνουμε έναν αλγόριθμο συμπίεσης κόμβων που ελαχιστοποιεί τη χρήση μνήμης, ενώ διατηρεί υψηλή απόδοση.

Συνοψίζοντας, η παρούσα διατριβή προσφέρει ένα ολοκληρωμένο πλαίσιο για τη βελτίωση της απόδοσης στις σχεσιακές, χρονικές και χωρικές βάσεις δεδομένων. Εστιάζει στη δημιουργία ευρετηρίων για την κύρια μνήμη, στη χρήση τεχνικών παράλληλης επεξεργασίας και στην αποφυγή διπλοτύπων, προσφέροντας αποδοτικές λύσεις για τις συνεχώς εξελισσόμενες ανάγκες των σύγχρονων εφαρμογών.

# Chapter 1

# Introduction



Databases [1, 2] are designed to store, organize, and manage large amounts of data. In today's world, they play a central role in nearly every application, ranging from online banking systems to social media platforms, healthcare records, and scientific research. They allow data to be stored in a structured, efficient way, making it easy to retrieve and manipulate when needed. The core idea of a database is that it helps users and applications to interact with data in a consistent and reliable manner, ensuring that the information is organized, accessible, and up to date.

Data management lies at the core of databases, which are categorized into various types based on their structure, functionality, and approaches to storing and accessing data. Relational databases organize data into structured tables of rows and columns, making them ideal for structured data management. Temporal databases focus on tracking data changes over time, providing a history of records with time-related attributes. Spatial databases handle geographic and spatial data, storing information like coordinates and shapes for geographic information systems. Document databases store semi-structured data in formats like JSON or XML, offering management of

schema-less information. Graph databases are designed to model and analyze relationships between entities using nodes and edges, while key-value databases store data as simple key-value pairs, optimized for fast lookups and updates. Column-family databases, another type of non-relational database, organize data into flexible rows and dynamic columns, making them ideal for distributed systems handling large-scale data. On the other hand, relational databases rely on structured tables with fixed schemas. This dissertation focuses on relational, temporal, and spatial databases.

**Relational Databases.** The most common and traditional type of database is the relational database. In relational databases, data is stored in tables that consist of rows and columns. Each row represents an individual record, and each column holds a specific type of data related to that record. For instance, in a database used for managing customer information, one table might store customer names and contact details, and yet another might store transaction records. The strength of relational databases lies in their ability to represent complex relationships between different data sets by using foreign keys, enabling data to be linked across multiple tables. Users interact with relational databases using structured query language (SQL), a powerful tool for querying and manipulating data.

**Temporal Databases.** Temporal databases are designed to manage data that changes over time. They extend the functionality of relational systems by adding time-related attributes to records. This allows users to query past states of data, track changes over time, and plan for future states. For example, a temporal database could store information about a person's address history, tracking the changes over time along with the effective dates. Temporal data is essential in systems such as financial transactions, medical records, and historical data analysis, where knowing not just the current state of information but also its history is crucial.

**Spatial Databases.** Spatial databases, are designed to handle data related to geographic locations, such as coordinates, maps, and geometric shapes. Spatial databases are commonly utilized in applications that involve geographic information systems (GIS), location-based services, and mapping technologies. These databases allow users to store, query, and analyze spatial data, such as the location of buildings, roads, or natural features. The data in spatial databases is often stored in forms that can represent points, lines, and polygons, making it particularly useful for any application

that needs to process and interpret spatial relationships and patterns.

**In-Memory Indexing.** While relational, temporal, and spatial databases each focus on different kinds of data, they all share the common challenge of efficiently storing and retrieving data. One key element that significantly impacts the performance of any database system is how quickly data can be located and retrieved from storage. This is where indexing becomes important. Indexing is a technique used to speed up the retrieval of data by creating data structures that allow for fast searches. Indices also help with other tasks in the database, like sorting data, combining information from different tables, and making sure that certain data, like unique IDs, does not repeat. Indexing is especially helpful when a query is searching for specific information or organizing large sets of data.

In recent years, a growing focus has been placed on in-memory indexing, which takes advantage of the computer's RAM rather than relying solely on slower disk storage. By storing indices in memory, the database can achieve much faster query response times, making it a critical aspect of performance for modern applications. In-memory databases and indexing methods have been especially useful in scenarios that require real-time data processing, such as high-frequency trading, gaming, and online recommendation systems. In-memory indexing is supported by efficient algorithms and data structures that optimize memory usage and access speed.

In relational databases, in-memory indexing can greatly improve performance by reducing the time needed to find and retrieve specific rows or values. Methods like hash indexing, bitmap indexing, and tree-based indexing (such as $B^+$-tree) are commonly used to build fast, in-memory search structures that speed up queries.

Temporal databases, must handle data that changes over time. This means they need indexing methods that can manage time-related queries efficiently, such as finding records valid at a specific time or retrieving historical data over a period. Techniques like time-based partitioning and versioned indexing help make these time-based queries faster and more efficient.

Spatial databases face unique challenges due to the nature of spatial data, which includes multidimensional points (such as latitude and longitude) and complex shapes. Unlike regular data, spatial data need specialized indexing methods like R-trees[3]. These structures organize data hierarchically to quickly filter out irrelevant information. They are especially useful for tasks like range queries (e.g., finding all locations within a certain distance) or spatial joins (e.g., combining data from different regions

based on their locations).

**Parallelism.** As the demand for faster data retrieval grows, especially in large-scale database systems, traditional indexing techniques alone are often not sufficient to meet the performance requirements of modern applications. In-memory indexing significantly improves query response times by minimizing dependence on slower disk storage. However, even in-memory systems face challenges with handling large datasets or complex queries.

To address these challenges, parallel programming models are being used in indexing algorithms to improve performance. There are many parallel programming models, such as shared memory, distributed memory, data parallelism, task parallelism, MapReduce, and others. In this study, we focus on the shared memory model and data parallelism.

The shared memory model allows multiple threads or processes to share the same memory space, making data sharing more efficient. Tools like mutexes or semaphores are needed to prevent conflicts and ensure safety when accessing shared data. This model is commonly used for parallel computing on multicore processors and is supported by frameworks like OpenMP and POSIX Threads. In this study, we use OpenMP [4] as the primary framework for implementing the shared memory model. OpenMP offers a simple and flexible interface for managing thread-level parallelism, making it an excellent choice for optimizing indexing operations on multicore systems. The data parallelism model performs the same operation on different parts of a dataset simultaneously, distributing the workload across multiple processors or threads. It is ideal for tasks like numerical computations, matrix operations, and vector processing. Data parallelism relies on multiple mechanisms, including SIMD (Single Instruction, Multiple Data), multithreading, distributed systems, GPUs, FPGAs, and big data frameworks. In this study, we focus on utilizing SIMD [5] instructions, which enable the simultaneous processing of multiple data elements with a single operation. By leveraging SIMD, we aim to optimize computational tasks such as vector processing, array operations, and other parallel workloads critical to our indexing algorithms. By distributing indexing tasks across multiple cores, OpenMP enables the efficient use of multi-core systems, significantly speeding up operations such as index construction and search queries. SIMD, on the other hand, allows for simultaneous processing of multiple data elements in a single instruction, optimizing computational tasks like comparisons or sorting within an index.

Combining the above parallel programming models with in-memory indexing, we can create faster and more efficient indexing structures that help databases scale better and handle larger datasets with reduced latency.

This dissertation focuses on in-memory parallel indexing techniques for temporal (Section 1.1), spatial (Section 1.2), and relational (Section 1.3) databases. In more detail, Section 1.1 introduces three different strategies for parallel partitioning of interval data, applicable to both hash-based and domain-based methods, which improve partitioning speed by reducing computation time and increasing scalability. Section 1.2 discusses challenges in spatial databases. First, we look at improving the Partition-Based Spatial Join (PBSM) algorithm for parallel in-memory spatial intersection joins. Then, we explore how different partitioning parameters affect the performance of spatial joins. We also focus on how to divide spatial space into a grid where the partitions can be processed in parallel without duplication, particularly for spatial intersection joins and range queries (rectangle-shaped and disk-shaped). Finally, Section 1.3 presents the problem of improving the performance of a $B^+$-tree in relational databases by applying parallelism and other techniques.

## 1.1 Parallel Partitioning In-Memory for Interval Joins

Generally speaking, temporal databases store relations of explicit attributes that conform to a schema and each tuple carries a *validity interval*. The interval join is one of the most widely used operations in temporal databases [6]. In this context, an interval join would find pairs of tuples from two relations which have intersecting validity. For example, assume that the employees of a company may be employed at different departments during different time periods. Given the employees in Figure 1.1 who have worked in departments A (red), B (blue), the interval join would find pairs of employees, whose periods of work in A and B, respectively, overlap.

Given a 1D discrete or continuous domain, an interval is defined by a starting and an ending point in this domain. Consider for example the domain of all non-negative integers $\mathbb{N}$; two integers $start, end \in \mathbb{N}$, with $start \leq end$ define an interval $i = [start, end]$ as the subset of $\mathbb{N}$, which includes all integers $x$ with $start \leq x \leq end$.[1] Let $R$, $S$ be two collections of intervals. Formally the *interval join* $R \bowtie S$ is defined

[1]Note that the intervals in this work are *closed*. Yet, our techniques and discussions apply on generic intervals where the begin and end sides are either open or closed.

| employee | start | end |
|---|---|---|
| John | 1994 | 2002 |
| Mary | 1992 | 2006 |

| employee | start | end |
|---|---|---|
| Jane | 1990 | 1993 |
| Bob | 1995 | 1996 |
| Hugo | 1997 | 2003 |
| Helen | 2005 | 2007 |
| Tom | 2006 | 2008 |

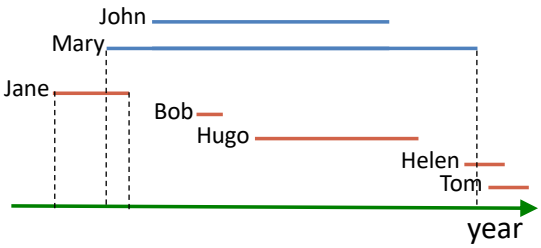



Figure 1.1: Motivation example in temporal databases

by all pairs of intervals $r \in R$, $s \in S$ that *intersect*, i.e., $r.start \leq s.start \leq r.end$ or $s.start \leq r.start \leq s.end$. My work focuses on data partitioning, such that the join can be processes in parallel and independently at each partition.

**Contributions.** The current state of the commodity hardware, having relatively large memory and the ability of parallel multi-core processing, motivated us to design novel techniques for partitioning the data. We investigate three different parallel strategies for the partitioning phase (applicable on both the hash-based and the domain-based partitioning), showing how it can benefit from modern hardware. These three strategies are implemented and tested.

## 1.2 Parallel In-Memory Evaluation of Spatial Queries

In this section, we introduce the concept of spatial queries and their importance in modern data processing. In Section 1.2.1, we present our contributions in spatial joins by improving the well-known Partition-Based Spatial Join (PBSM) algorithm. We explore how adjusting partitioning parameters can lead to significant improvements in the efficiency and speed of join queries, providing a more effective approach for handling large-scale spatial data. In Section 1.2.2, we focus on the challenge of indexing non-point data, aiming to enhance the performance and efficiency of spatial queries. By optimizing the indexing process, we seek to reduce computational overhead. Additionally, we focus on preventing the creation of duplicate results, which can slow down query execution.

### 1.2.1 Parallel In-Memory Evaluation of Spatial Joins

The spatial join is a well-studied fundamental operation, that finds application in spatial database systems [7] and Geographic Information Systems (GIS) [8]. GIS, for example, typically store multiple thematic layers (e.g., road network, hydrography), which are spatially joined in order to find object pairs (e.g., roads and rivers) that intersect. Besides, spatial joins are also used to support data mining operations such as clustering [9] and pattern detection [10].

Given two collections of spatial objects $R$ and $S$, the *spatial intersection join* returns all $(r, s)$ pairs, such that $r \in R$, $s \in S$ and $r$ and $s$ have at least one common point. Due to the potentially complex geometry of the objects, intersection joins are typically processed in two steps. The *filter* step applies on spatial approximations of the objects, typically their *minimum bounding rectangle* (MBR). For each pair of object MBRs that intersect, the object geometries are fetched and compared in a *refinement step*. Similar to the vast majority of previous work [11], we focus on the filter step.

A wide range of spatial join algorithms have been proposed in the literature [12]. Most of them assume that the input data are disk-based and their objective is to minimize I/O accesses during the join. Given the fact that main memory chips become bigger and faster, in-memory join processing has recently received a lot of attendance [13]. In addition, given that commodity hardware supports parallel processing, multi-core join evaluation has also been the focus of recent research. Hence, we target the parallel in-memory evaluation of spatial joins on modern hardware.

Our focus is the optimization of the simple, but powerful partitioning-based spatial join (PBSM) algorithm [14]. PBSM is shown to perform well in previous studies [13] and used by most distributed spatial data management systems [15, 16, 17]. In a nutshell, both datasets are first partitioned using a regular grid; each tile (cell) of the grid gets all rectangles that intersect it. The (possibly very large number of) tiles are grouped into a smaller number of partitions in a round-robin fashion according to their z-ordering [15]. Each tile defines a smaller spatial join task. These tasks are independent and can be executed in parallel, assigned to different threads or even to different machines in distributed evaluation. Typically a plane sweep algorithm based on forward scans [18] is used to process each task. For example, consider the two sets of MBRs of Figure 1.2a. Partitioning the rectangles using a $3 \times 3$ grid creates 9 independent spatial join tasks, one for each tile. Note that some rectangles may be

replicated to multiple tiles. Because of this, some pairs of rectangles may be found to intersect in multiple tiles; e.g., $r_1$ intersects $s_1$ in tiles (0,0) and (0,1).

The classic approach to eliminate duplicates is to hash the query results and identify duplicates at each bucket. This method is very expensive, especially when the number of results is large. An improved hashing technique for spatial data that limits the size of the hash table was proposed in [19]. The state-of-the-art technique for duplicate elimination, used in most big spatial data management systems [20], is the *reference point* approach [21]. Using the reference point approach, duplicate join results can be avoided by reporting a pair of rectangles only if a pre-determined reference point (typically, the top-left corner) of the intersection region is in the tile [21]; e.g., $(r_1, s_1)$ is only reported by tile (0,0).

**Contributions.** Currently, there is no comprehensive study so far on how the number and type of partitions should be defined. We observed, by experimentation, that changing the type and the number of partitions can make a big difference. At first, we evaluate a 1D partitioning that divides the space into stripes (see Figure 1.2b), as opposed to the classic 2D partitioning, which uses a grid. Further, we investigate, for each partition, the best direction of the sweep line. Finally, we show how both the partitioning and the joining phases of the algorithm can be parallelized. Based on our experimental findings, the 1D partitioning results in a more efficient algorithm. Also, increasing the number of partitions improves the performance of the algorithm, up to a point where adding more partitions starts having a negative effect. We present a number of empirical rules driven from data statistics (globally and locally for each partition) that can guide the selection of the algorithm's parameters. Finally, we evaluate the performance of the parallel version of the algorithm and show that it scales gracefully with the number of cores.

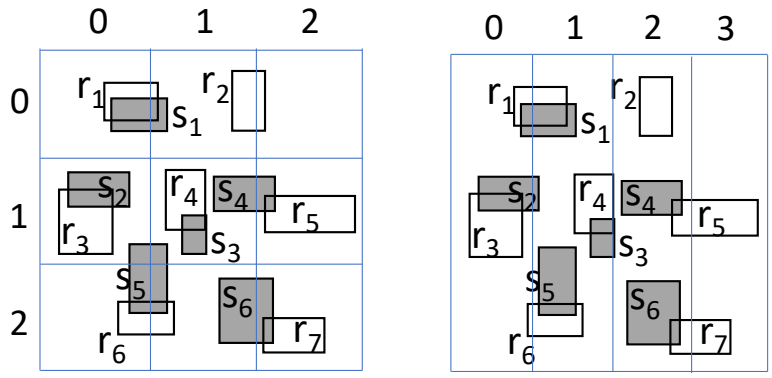



Figure 1.2: Example of PBSM: (a) 2D and (b) 1D partitioning

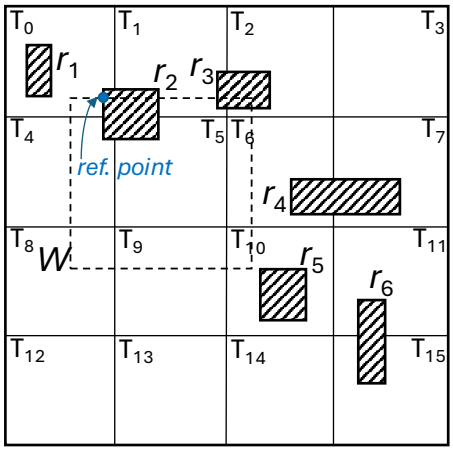
### 1.2.2 A Two-layer Partitioning for Non-point Spatial Data

Spatial data management has been studied for decades [22]. With modern, affordable large memories and multi-core processors enabling parallel query processing, spatial object collections can now fit in the memory of commodity machines. However, despite advancements in distributed systems for spatial data [16, 17, 23, 24, 20], in-memory management of large-scale spatial data remains under-explored.

We investigate the problem of indexing non-point spatial objects (e.g., polygons, linestrings, etc.) in memory, for the efficient single- and multi-threaded evaluation of spatial range queries. Large volumes of non-point data are ubiquitous, hence, their effective management is always timely. Besides Geographic Information Systems, domains that manage big volumes of such data include graphics (e.g., management of huge meshes [25]), neuroscience (e.g., building and indexing a spatial model of the brain [26]), and location-based analytics (e.g., managing spatial influence regions of mobile users in order to facilitate effective POI recommendations [27]).

**Motivation.** Spatial access methods can be divided into two categories; *space-oriented partitioning* (SOP) and *data-oriented partitioning* (DOP) approaches. Indices of the first category divide the space into *spatially disjoint* partitions. As a result, objects that overlap with multiple partitions need to be replicated (or clipped) in each of them. DOP methods allow the extents of the partitions to overlap and ensure that their contents are disjoint (i.e., each object is assigned to exactly one partition). For disk-resident data, DOP approaches (such as the R-tree [3] and its variants) are considered to be the best, because they avoid data replication and they have a balanced structure. However, SOP approaches (especially grids) are gaining ground due to their efficiency in search and updates in main memory [28, 29, 30, 31, 32, 33]. In addition, query evaluation over grids is embarrassingly parallelizable and SOP is widely used in distributed spatial data management systems [17, 23, 24].

We focus on improving SOP indices by addressing an inherent problem they have: potential duplicate query results. In particular, a range query may overlap multiple partitions which may include multiple replicas of the same object. For example, consider the six rectangular objects depicted in Figure 1.3, partitioned using a $4\times4$ grid. Some objects are assigned to multiple tiles. Given a query range (e.g., $W$), a replicated object (e.g., $r_2$) may be identified as query result multiple times (e.g., at tiles $T_0$, $T_1$, $T_4$, and $T_5$). Using the state-of-the-art reference point approach for duplicate elimination,




| Tile | primary partitioning | secondary partitioning |
|---|---|---|
| $T_0$ | $\{r_1, r_2\}$ | A = $\{r_1, r_2\}$ |
| $T_1$ | $\{r_2, r_3\}$ | A = $\{r_3\}$, C = $\{r_2\}$ |
| $T_2$ | $\{r_3\}$ | C = $\{r_3\}$ |
| $T_4$ | $\{r_2\}$ | B = $\{r_2\}$ |
| $T_5$ | $\{r_2\}$ | D = $\{r_2\}$ |
| $T_6$ | $\{r_4\}$ | A = $\{r_4\}$ |
| $T_7$ | $\{r_4\}$ | C = $\{r_4\}$ |
| $T_{10}$ | $\{r_5\}$ | A = $\{r_5\}$ |
| $T_{11}$ | $\{r_6\}$ | A = $\{r_6\}$ |
| $T_{15}$ | $\{r_6\}$ | B = $\{r_6\}$ |

Figure 1.3: Example of partitioning and object classes

we can avoid duplicates in range queries. More specifically, for each query result $r_i$, found in a tile $T$, this approach computes a reference point of the intersection between $r_i$ and the query window $W$ (e.g., the upper-left corner in Fig. 1.3). If the reference point is inside $T$, then $r_i$ is reported, otherwise it is ignored. Since the reference point can only be inside one tile, no duplicate results are reported. Although this method avoids hashing, we still have to bear the cost of retrieving duplicate copies of the same object and computing the reference point for each copy.

We propose a secondary partitioning technique for SOP indices, which improves their performance significantly, by avoiding the generation and elimination of duplicate results. Our approach is novel and of a high impact, as (i) it is extremely easy to implement, (ii) it can be used by any SOP index, and (iii) it can be directly implemented in big spatial data management systems [20]. In a nutshell, we divide the objects which are assigned to each partition $T$ into four classes $A, B, C, D$. Objects in class $A$ begin inside $T$ in both dimensions, objects in class $B$ start inside $T$ in dimension $x$ only, objects in class $C$ start inside $T$ in dimension $y$ only, and objects in class $D$ start before $T$ in both dimensions. Fig. 1.3 exemplifies how the objects are divided into classes. For example, in tile $T_1$, object $r_2$ belongs to class $C$, because $r_2$ starts before $T_1$ in the $x$ dimension and starts inside $T$ in the $y$ dimension. During query evaluation, for each partition $T$ which intersects the query range, we access *only* the object classes in $T$ that are guaranteed not to produce duplicate results. For example, in tile $T_1$ of Fig. 1.3, we will not access class $C$, because query $W$ starts before $T_1$ in dimension $x$; i.e., any object in class $C$ of $T_1$ that intersects $W$ should also intersect $W$ in the previous tile $T_0$. Hence, we avoid verifying whether $r_2$ intersects $W$ before

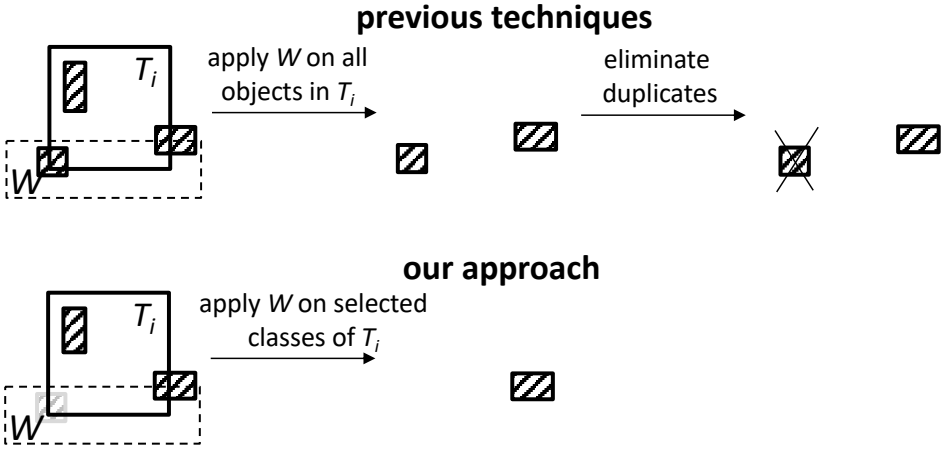


Figure 1.4: Comparison between our approach and previous work

realizing that it is a duplicate result. Object $r_2$ will only be accessed in tile $T_0$ and reported as result there without any duplicate check.

Then, we explain in detail how range queries are evaluated by our scheme and show how redundant computations and duplicate checks can be avoided overall. Fig. 1.4 illustrates the difference between our approach and the deduplication process followed by previous work [21, 19]; while all previous approaches evaluate queries on *all* objects of each partition and then eliminate possible duplicates, we process only a subset of objects in each partition that cannot be duplicates and we do not perform any deduplication.

Besides proposing a secondary partitioning technique for duplicate avoidance in range queries, we show how to reduce the number of required comparisons per rectangle to at most one per dimension. Furthermore, we propose a data decomposition approach which further reduces the number of comparisons We also study the evaluation of circular range (i.e., disk) queries and, in general, queries with convex range shapes. Next, we focus on non-rectangular objects, which are approximated and indexed using their minimum bounding rectangles (MBRs). Moreober, we show that for such objects, the expensive query refinement step in query evaluation can be avoided in most cases by a simple post-filtering test on the object MBRs. Additionally, we investigate the evaluation of multiple range queries in batch and in parallel, using our secondary partitioning approach.

Afterward, we turn our focus to spatial intersection joins. We first discuss join evaluation for two datasets that are primarily indexed by identical grids. To avoid duplicate results, we show that, for each tile, it suffices to evaluate 9 out of the 16 possible joins between the pairs of secondary partitions in the tile. We also show

how to optimize the join phase of PBSM, by specialized plane-sweep routines for the different cases of joined sub-partitions (classes) and by avoidance of redundant comparisons. Finally, we investigate join evaluation when one or both joined inputs have already been indexed and how to process joins of datasets that have been partitioned using a different grid.

Finally, we evaluate our proposal experimentally using large publicly available real datasets and synthetic ones of the same scale as those used in recent work [20, 34, 35]. Our experiments (with workloads of queries and updates) show that main-memory grids are superior to alternative SOP and state-of-the-art DOP indices, which justifies our focus to improve SOP indexing. More importantly, we show that when we replace the state-of-the-art duplicate elimination technique [21] by our secondary partitioning technique, the performance of grid-based indexing is improved by up to a few times. Overall, a grid index equipped with our secondary partitioning technique is up to one order of magnitude faster compared to the best performing DOP index (an in-memory R-tree implementation from boost.org) for range queries of varying sizes, achieving an impressive throughput of tens of thousands of queries per second. We also show that our (directly parallelizable) approach scales gracefully with the number of cores (i.e., threads in a multi-core machine), making it especially suitable for shared-nothing parallel environments where tree-based indices are hard to deploy. Finally, we demonstrate that in-memory spatial indexing can be orders of magnitude faster compared to distributed spatial data management systems for the scale of data used in our experiments.

**Contributions.** In summary, we make the following contributions to spatial indexing and query evaluation. We design a novel second-layer partitioning approach designed for space-oriented partitioning (SOP) indices, such as grids. Our approach enhances SOP indices by avoiding redundant object comparisons and the generation of duplicate results, while also minimizing the cost of intersection tests. Additionally, we propose an efficient filtering mechanism that largely eliminates the need for refinement steps, significantly improving the performance of range queries. We extend this methodology to support the evaluation of multiple range queries in both batch and parallel processing scenarios. Furthermore, we adapt the second-layer partitioning approach for spatial joins, presenting new joining strategies and performance optimizations. An extensive experimental evaluation demonstrates the superiority of SOP indices over direction-oriented partitioning (DOP) indices, as well as the advantages

of our partitioning approach over the state-of-the-art duplicate elimination technique [21].

## 1.3 $B^S$-tree: A data-parallel $B^+$-tree for main memory

Building on our previous work in parallel indexing for in-memory spatial and interval data, we decided to focus on a more traditional indexing structure for relational data. Specifically, we explored the $B^+$-tree structure to enhance it using modern hardware capabilities and new techniques, aiming to improve its performance and efficiency. Another reason for this choice is the recent development of learned indices, which use machine learning models to "learn" the data distribution. These indices are claimed to be more efficient in terms of performance and memory usage compared to traditional $B^+$-trees. However, we believe that an optimized $B^+$-tree, designed to leverage modern hardware, can outperform learned indices while offering greater stability. Unlike learned indices, which can vary in performance depending on the data distribution, $B^+$-trees deliver consistent behavior regardless of the dataset.

$B^+$-trees have already proven highly effective for various types of data, including temporal and spatial. Their design, which keeps data sorted, makes them particularly suitable for different types of queries. For example, range queries in temporal data, such as "Find all events that occurred within a specific time range," can benefit from using timestamps as keys in a $B^+$-tree. Similarly, for spatial data, queries like "Retrieve all points within a specific rectangular region" can also be supported efficiently. To handle multi-dimensional spatial data, techniques like Z-ordering or Hilbert curves can be used to transform multi-dimensional points into a single-dimensional space, making them compatible with $B^+$-tree indexing. Moreover, for spatio-temporal data, such as trajectories, indexing the time dimension alongside spatial data can further enhance query performance, demonstrating the $B^+$-tree 's versatility and effectiveness.

The $B^+$-tree has been the dominant indexing method for DBMSs, due to its low and guaranteed cost for query processing and updates and due to its support of range queries (in addition to equality searches, which are also well-supported by hash indices). It has been designed for disk-based storage, where the objective is to minimize the I/O cost of operations. As memories become larger and cheaper, main-memory and hardware-specific implementations of the $B^+$-tree [36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47], as well as alternative access methods for in-memory data

[48, 49, 50, 51, 52, 53, 54, 55, 56] have been proposed. The optimization objective in all these methods is minimizing the computational cost and cache misses during search. More recently, learned indices [57, 58, 59, 60, 61, 62, 63, 64, 65, 66], which replace the inner nodes of the B$^+$-tree by ML models have been suggested as a way for reducing the memory footprint of indexing and accelerating search at the same time. Cache conscious B-trees and other data structures have also been developed.

We propose B$^S$-tree, a B$^+$-tree for main memory data, which is optimized for modern commodity hardware and data parallelism. B$^S$-tree adopts the structure of the disk-based B$^+$-tree (i.e., a balanced, multiway tree), setting the node size to a memory block that can be processed fast. At the heart of our proposal lies a data-parallel *successor* operator ($succ$), implemented using SIMD, which is applied at each tree level for branching during search and updates and for locating the search key position at the leaf level. To facilitate fast updates, we propose a novel implementation for gaps (unused positions) by duplicating keys, that does not affect the excellent search performance of the tree. The B$^+$-tree construction algorithm initializes sparse leaf nodes with intentional gaps in them, in order to (i) delay possible splits and (ii) reduce data shifting at insertions. The splitting algorithm also adds gaps proactively. Finally, we propose a compression method that allows nodes that use fixed-size memory blocks to have varying capacities, which saves space and increases data parallelism. The computational cost of search and update operations in B$^S$-tree is $O(\log_f n)$, where $f$ is the capacity of the nodes, assuming that $f$ is selected such that each node can be processed by a (small) constant number of SIMD instructions.

**Contributions.** There already exist several SIMD-based implementations of B-trees and k-ary search [67, 41, 45, 46, 47]. In addition, several indices (especially learned ones [60]) use gaps to facilitate fast updates. Finally, key compression in B-trees has also been studied in previous work [40]. To our knowledge, our proposed B$^S$-tree is the first structure that gracefully combines all these features, achieving at the same time minimal storage and high throughput. All these thanks to (i) our simple but efficient data-parallel implementation of branching; (ii) its integration with a novel implementation of gaps using duplicate keys that does not affect correctness and performance; and (iii) our compression scheme that allows for direct application of operations on compressed nodes. We extensively compare our B$^S$-tree implementation with open-source implementations of state-of-the-art non-learned and learned indices on widely used real datasets, to find that B$^S$-tree and its compressed version

consistently prevails in different query and update workloads, typically achieving 1.5x-2x higher throughput than the best competitor from previous work.

## 1.4 Dissertation Outline

The rest of this dissertation is organized as follows. In Chapter 2, we review related work and present in detail the characteristics and weaknesses of existing methods.

In Chapter 3, we present our parallel strategies, designed specifically for efficiently partitioning intervals. These strategies are applicable to both hash-based and domain-based partitioning algorithms. Our goal was to identify the most optimal approach based on performance metrics, ensuring that the proposed methods are both practical and scalable for real-world applications.

In Chapter 4 we present our contribution to spatial indexing and query evaluation. Our first study focused on the in-memory and parallel evaluation of spatial joins by improving the classic PBSM algorithm using one-dimensional (stripes) and two-dimensional (grid) partitioning methods. We also explain how to choose the right partitioning parameters based on data statistics to optimize the algorithm for specific join inputs. Next, we introduce a new secondary partitioning technique for space-based partitioning indices (e.g., grids). This technique greatly improves performance by avoiding duplicate results and can be used for range queries (such as rectangles and disks) and spatial intersection join queries. Finally, we conducted a wide range of experiments using both real and synthetic datasets to validate our techniques

In Chapter 5, we propose $B^S$-tree, an in-memory $B^+$-tree optimized for fast, parallel processing using SIMD instructions. When compared to existing main-memory and learned indices, our approach demonstrates superior performance across a range of query and update workloads.

In conclusion, Chapter 6 summarizes the contributions of this dissertation and provides a discussion about future work.

# Chapter 2

# Related Work



In this chapter, we will discuss the related work for this dissertation, focusing on previous research in interval joins (Section 2.1), spatial data management (Section 2.2), and relational data, particularly tree-like structures (Section 2.3).

## 2.1 Interval Joins

Recall that the objective of the interval join is to find pairs of intervals that overlap each other. The majority of interval join algorithms assume that the input data reside on disk and so, their focus is to minimize I/O accesses.

In this section, we classify the algorithms of previous works based on the data structures they use and based on the underlying architecture.

**Nested loops and merge join.** Early work on interval joins [68, 69] studied a temporal join problem, where two relations are equi-joined on a non-temporal attribute and the temporal overlaps of joined tuple pairs should also be identified. Techniques based on nested-loops (for unordered inputs) and on sort-merge join (for ordered inputs) were proposed, as well as specialized data structures for append-only databases. Similar to

plane sweep, merge join algorithms require the two input collections to be sorted, but join computation is sub-optimal compared to FS, which guarantees at most $|R| + |S|$ endpoint comparisons that do not produce results.

**Index-based algorithms**. Enderle et al. [70] propose interval join algorithms, which operate on two RI-trees [71] that index the input collections. Zhang et al. [72] focus on finding pairs of records in a temporal database that intersect in the (key, time) space (i.e., a problem similar to that studied in [68, 69]), proposing an extension of the multi-version B-tree [73].

**Partitioning-based algorithms**. A partitioning-based approach for interval joins was proposed in [74]. The domain is split into disjoint ranges. Each interval is assigned to the partition corresponding to the last domain range it overlaps. The domain ranges are processed sequentially from last to first; after the last pair of partitions are processed, the intervals which overlap the previous domain range are *migrated* to the next join. This way data replication is avoided. Histogram-based techniques for defining good partition boundaries were proposed in [75]. A more sophisticated partitioning approach, called Overlap Interval Partitioning (OIP) Join [76], divides the domain into equal-sized granules and consecutive granules define the ranges of the partitions. Each interval is assigned to the partition corresponding to the smallest sequence of granules that contains it. In the join phase, partitions of one collection are joined with their overlapping partitions from the other collection. OIP was shown to be superior compared to index-based approaches [70] and sort-merge join. These results are consistent with the comparative study of [6], which shows that partitioning-based methods are superior to nested loops and merge join approaches.

Disjoint Interval Partitioning (DIP) [77] was recently proposed for temporal joins and other sort-based operations on interval data (e.g, temporal aggregation). The main idea behind DIP is to divide each of the two input relations into partitions, such that each partition contains only disjoint intervals. Every partition of one input is then joined with all of the other. Since intervals in the same partition do not overlap, sort-merge computations are performed without backtracking. Prior to this work, temporal aggregation was studied in [78]. Given a large collection of intervals (possibly associated with values), the objective is to compute an aggregate (e.g., count the valid intervals) at all points in time. An algorithm was proposed in [78] which divides the domain into partitions (buckets), assigns the intervals to the first and last bucket they

overlap and maintains a meta-array structure for the aggregates of buckets entirely covered by intervals. The aggregation can then be processed independently for each bucket (e.g., using a sort-merge based approach) and the algorithm can be parallelized in a shared-nothing architecture. We also propose a domain-partitioning approach for parallel processing (Section 3.1), but the details differ due to the different natures of temporal join and aggregation. Yet another partitioning approach [79] models each interval $r$ as a 2D point ($r$.start,$r$.end) and divides the points into spatial regions. Again, a partition of one collection should be joined with multiple partitions of the other collection.

**Methods based on plane sweep**. The Endpoint-Based Interval (EBI) Join [80] is the most recent approach and we consider it to be the state-of-the-art. EBI is an efficient implementation of plane sweep, which is based on a specialized *gapless hash map* data structure for managing the active sets of intervals. EBI and its lazy version LEBI were shown to significantly outperform OIP [76] and to also be superior to another plane sweep implementation [81]. An approach similar to EBI is used in SAP HANA [82]. To our knowledge, no previous work was compared to FS [18]. Last, extensions and applications of the plane sweep approach has been discussed in [83, 84], but in the context of temporal aggregation and SPARQL query processing, respectively.

**Parallel algorithms**. A domain-based partitioning strategy for interval joins on multi-processor machines was proposed in [85]. An interval is assigned to the partition corresponding to the domain interval where its start endpoint belongs. Each partition is assigned to a processor and intervals are replicated to the partitions they overlap, to allow join results being produced independently at each processor. At the end, a merge phase with duplicate elimination is required as the same join result can be produced by different processors. Duplicates can be avoided using the reference test from [21] but, this approach incurs extra comparisons.

**Distributed algorithms**. Distributed interval joins evaluation was studied in [86]. The goal is to join sets of intervals, which are located at different clients. The clients iteratively exchange statistics with the server, which help the latter to compute a coarse-level approximate join; exact results are refined by on-demand communication with the clients. Chawda et al. [87] implement the partitioning algorithm of [85] in the MapReduce framework and extend it to operate for other (non-overlap) join predicates. The main goal of distributed algorithms is to minimize the communication

cost between the machines that hold the data and compute the join.

## 2.2 Management of Spatial Data

### 2.2.1 Indexing Non-point Spatial Objects

The goal of spatial indices (memory resident or disk-based) is to group closely located objects in space, into the same index nodes or blocks. These blocks are then organized in a (single-level or hierarchical) data structure. The first spatial indices were designed for point data, which are easier to manage. Later, the focus shifted to non-point data, which are harder to manage. Spatial queries are typically processed in two steps [22], following a *filtering-and-refinement* framework. During *filtering*, the query is applied on the MBRs, which approximate the objects. During *refinement*, the exact representations of the candidates are accessed and tested against the query predicate. Spatial indices are applied in the filtering step; hence, they manage MBRs instead of exact geometries.

Depending on the nature of the partitioning, spatial indices are classified into two classes [88]. Indices based on *Space-oriented partitioning* (SOP) divide the space into disjoint partitions and were originally designed for point data. A grid [89], which divides the space into cells (partitions) using axis-parallel lines, is the simplest SOP index. Hierarchical indices that fall in this category are the kd-tree [90] and the quad-tree [91],[92]. A bitmap-based index for point data was recently proposed in [93]. SIDI [94] is another spatial index for point data, which learns the characteristics of the dataset before construction and its layout is designed to fit the data well. SOP can also be used for non-point objects; in this case, objects whose extent overlaps with multiple partitions are replicated (or clipped) in each of them [95].

Due to object replication, the same query results may be detected in multiple partitions and deduplication techniques should be applied. Aref and Samet [19] improve the baseline hash-based duplicate elimination technique by processing the partitions in a specific order, which guarantees that duplicates may appear only in a subset of partitions (called *active border*). The size of the active border determines the space requirements of the hash table. The state-of-the-art deduplication technique by Dittrich and Seeger [21] avoids the use of a hash table and performs a simple check for each produced result. It computes a *reference point* of the intersection area between each result and the query range. If the reference point is inside the partition, then

the result is reported, otherwise it is eliminated as a duplicate.

Indices based on *data-oriented partitioning* (DOP) allow the extents of the partitions to overlap and ensure that their contents are disjoint (i.e., each object is assigned to exactly one partition); hence, there is no need for result deduplication. Variants of the R-tree [3] (e.g., the R*-tree [96]) are the most popular methods in this class. The R-tree is a height-balanced tree, which generalizes the $B^+$-tree in the multi-dimensional space and hierarchically groups object MBRs to blocks. Each block is also approximated by an MBR, hence the tree defines a hierarchy of MBR groups. Some R-tree variants use circles (or spheres in the 3D space) instead of MBRs, i.e., the SS-tree [97], or a combination of circles and rectangles, i.e., the SR-tree [98]. The R-tree was originally proposed for disk-resident data and the key focus is minimizing the I/O cost during query processing. The CR-tree [99] is an optimized R-tree for the memory hierarchy. BLOCK [88] is a recently proposed main-memory DOP index, which uses a hierarchy of grids. R*-Grove [100] is a spatial partitioning, which builds on the split algorithm of R*-tree to define full blocks and balanced partitions for distributed big data.

Recently, following the trend for relational data, *learned indices* for spatial data have been proposed [101, 34, 35]. The main idea is to learn the spatial distribution of the objects, and then define a lightweight index, where search is guided by models instead of a sparse index. Based on this idea, Wang et al. [101] first map the data to a 1D space, using their Z-order, and then construct a multi-staged learned index for 1D data. In LISA [34], is a learned spatial index that focuses on disk-resident data; the data are organized using a grid; the 1D order of the cells and the data distribution determines the grouping of cells and the corresponding learned models. RSMI [35] suggests a rank space based ordering for point data, which becomes scalable by a recursive partitioning and learning strategy. These indices are not directly comparable to our work, because they are designed for point data (with no obvious extension to non-point data) and their primary goal is to minimize the I/O cost.

### 2.2.2 Range Queries

We now provide a more detailed explanation of how a range query can be evaluated using a simple grid.

Recall that each MBR $r$ can be represented by an interval of values at each di-

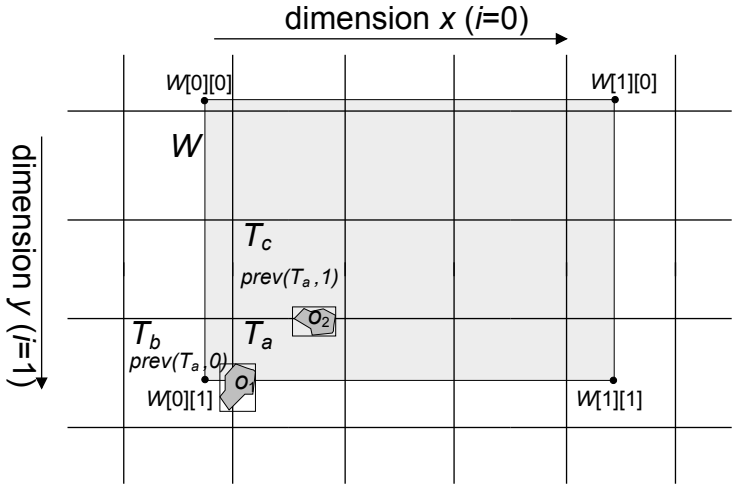



Figure 2.1: Example of tiling and query evaluation

mension. Let $r[i] = [r[i][0], r[i][1]]$ be the projection of rectangle $r$ on the $i$-th axis. For example, in the 2D space, $r[0][1]$ denotes the upper bound of rectangle $r$ on dimension $0$ (i.e., the $x$-axis). Similarly, we use $T[i] = [T[i][0], T[i][1]]$ to denote the projection of a tile $T$ to the $i$-th dimension. Given a tile $T$ and a dimension $i$, we use $prev(T, i)$ to denote the tile $T'$ which is right before $T$ in dimension $i$ and has exactly the same projection as $T$ in the other dimension(s). For example, in Figure 2.1, $T_b = prev(T_a, 0)$. $prev(T, i)$ is not defined for tiles $T$ which are in the first column (for $i = 0$) or row (for $i = 1$) of the grid.

Given a range query window $W$, a tile that does not intersect $W$ does not contribute any results to the query. Specifically, the only tiles $T$ that may contain query results are those for which $T[i][1] \geq W[i][0]$ and $T[i][0] \leq W[i][1]$ at every dimension $i$ and can easily be enumerated after finding the tiles $T_s$ and $T_e$, which contain $W[0][0]$ and $W[1][1]$, respectively.[1] Figure 2.1 illustrates a window query $W$ in lightgrey color and its four corner points $W[0][0]$, $W[0][1]$, $W[1][0]$, $W[1][1]$. The tiles which are relevant to $W$ are between (in both dimensions) the two tiles $T_s$ and $T_e$.[2]

For each tile which is totally covered by the query range in at least one dimension (e.g., $T_a$ in dimension $0$), we know that the objects in it certainly intersect $W$ in that dimension. For a tile $T$ that partially overlaps with $W$ in both dimensions (e.g., $T_b$), we need to iterate through its objects list to verify their intersection with $W$. We first check whether the MBR of the object intersects $W$ and then we might have to verify

[1] $T_s$ and $T_e$ can be found in $O(1)$ by algebraic calculations if the grid is uniform.

[2] We conventionally assume that the $x = 0$ dimension is from left to right and the $y = 1$ dimension is from top to bottom.

with the exact geometry of the object at a refinement step.

An important issue is that neighboring tiles may intersect $W$ and also contain the same object $o$. In this case, $o$ will be reported more than once, so we need an approach for handling these duplicates. For example, in Figure 2.1, object $o_1$ could be reported both by $T_a$ and by $T_b$. A solution to this problem is to report an object $o$ only at the tile which is before all tiles (in both dimensions) where $o$ is found to intersect $W$. For example, in Figure 2.1, $o_1$ is reported by $T_b$ only, which is before $T_a$. An easy approach to perform this test is to compute the intersection between the query window and the rectangle and report the result only if a *reference point* of the intersection (e.g., the smallest values of the intersection in all dimensions) is included in the tile [21]. While this solution prevents reporting duplicates, it requires extra computations and comparisons and it is unclear how to apply it for non-rectangular range queries. An alternative and more general (but more expensive) approach is to add the results from all tiles in a hash table, which would prevent the same rectangle from being added multiple times.

### 2.2.3 Spatial Joins

In this section, we review classic spatial join evaluation approaches and more recent work for in-memory. In general, in order to join spatially two large object collections $R$ and $S$, we first divide them into partitions which are small enough and then join the partitions. We may also exploit an existing partitioning or index. In either case, the join is broken down into numerous small problems that can be solved fast in memory. We first discuss how a (small) join problem can be processed in memory, using a plane sweep algorithm. Then, we review how data partitioning and indexing approaches can be used to process bigger spatial join problems.

#### 2.2.3.1 Evaluating Small Joins

For in-memory processing of small spatial joins, a typical approach is to use adaptations of a plane sweep algorithm that compute rectangle intersections [102]. The most commonly used adaptation was suggested by Brinkhoff et al. [18]. Algorithm 2.1 describes this method. The join inputs $R$ and $S$ are first sorted based on their lowest value in one dimension (e.g., $x_l$ of the $x$-dimension). Then, the sorted inputs are scanned concurrently and merged as in a merge-join. This resembles a line that

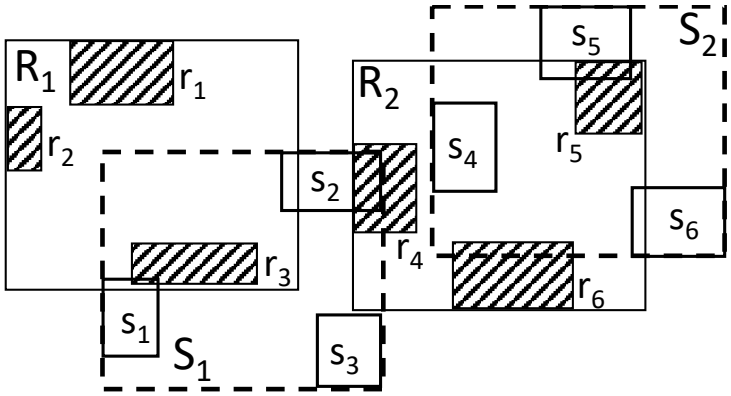
(is perpendicular to and) sweeps along the sorting dimension. For every value that the line encounters, say the lower $x$-endpoint $r.x_l$ of a rectangle $r \in R$, the other input, i.e., $S$, is *forwardly scanned* from the current rectangle $s' = s$, while $s'.x_l$ is *not greater than* the upper $x$-endpoint $r.x_u$ of $r$. All $s' \in S$ found in this scan are guaranteed to $x$-intersect $r$, so for each of them a $y$-intersection test is applied (Line 8) to confirm whether $r$ and $s'$ intersect. Arge et al. [81] studied more classic (but less simple to implement) versions of plane sweep based on maintenance of *active lists* at every position of the sweep line, which have insignificant performance differences to Algorithm 2.1.

#### 2.2.3.2 Data Partitioning

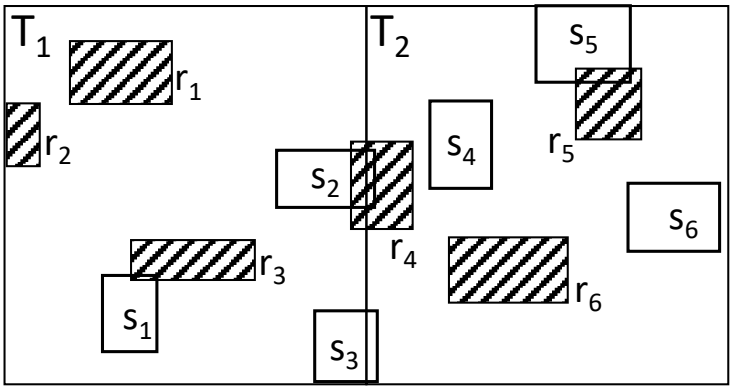
Spatially joining large inputs directly using Algorithm 2.1, without any preprocessing can be quite expensive. Some 20 years ago, the memories were too small to entirely fit the input data; hence, expensive sorting and sweeping would have to be performed in external memory. Given this, *data partitioning* has been considered as a divide-and-conquer approach which splits the two inputs into smaller subsets that can then be spatially joined fast in memory. In a nutshell, each object collection is divided into a number of partitions, such that objects that are spatially close to each other fall in the same partition. A partition from $R$ is then joined with a partition from $S$ if their MBRs intersect.

A large number of spatial join algorithms that follow this paradigm have been proposed. They can be classified into *single-assignment, multi-join* (SAMJ) methods and *multi-assignment, single-join* (MASJ) approaches [103]. SAMJ methods assign each object to exactly one partition; the partitions are determined by spatial clustering heuristics. A partition from one dataset (e.g., $R$) may have to be joined with multiple partitions of the other dataset (e.g., $S$). In MASJ, the borders of the partitions are predetermined, and an object is assigned to every partition it spatially intersects. Each partition from $R$ is then joined with exactly one partition from $S$ (which has exactly the same MBR). Figure 2.2 shows the differences between these two partitioning schemes. In SAMJ, illustrated in Figure 2.2(a), the (dark grey) rectangles of dataset $R$ are divided to partitions $R_1$ and $R_2$, while the (hollow) rectangles of $S$ are divided into groups $S_1$ and $S_2$. Partition $R_1$ only needs to be joined with $S_1$ because the MBR of $R_1$ does not intersect the MBR of $S_2$. However, $R_2$ should be joined with both $S_1$ and $S_2$. In MASJ, illustrated in Figure 2.2(a), the datasets are partitioned based on

the space division defined by tiles $T_1$ and $T_2$. The rectangles from $R$ that intersect a tile (e.g., $T_1$) only have to be joined with the rectangles from $S$ that are assigned to the same tile. Note that objects $\{r_4, s_2, s_3\}$, which intersect both tiles, are replicated.




(a) SAMJ




(b) MASJ

Figure 2.2: Two classes of partitioning techniques

A classic SAMJ approach, used when the two inputs are indexed by R-trees [3], is the *R-tree join* (RJ) algorithm of [18]. RJ finds all pairs of entries $(e_R, e_S)$ one from each root node of the trees that intersect. For each such pair, it recursively applies the same procedure for the nodes pointed by $e_R$ and $e_S$, until pairs of leaf node entries (which correspond to intersecting object MBRs) are found. For example, if $R_1$ and $R_2$ ($S_1$ and $S_2$, respectively) in Figure 2.2(a) are the two children of an R-tree root that indexes $R$ ($S$, respectively), then RJ would use their MBRs to determine that $R_1$ only needs to be joined with $S_1$. To join each pair of nodes, plane-sweep is used. RJ was later extended to a multiway join processing algorithm [104] that applies to multiple R-trees. Another SAMJ approach that does not rely on pre-defined indexes

is *Size Separation Spatial Join* [105].

The most popular MASJ approach is *Partition-based Spatial Merge Join* (PBSM) [14]. PBSM divides the space by a regular grid and objects from both join inputs are assigned to all tiles which spatially overlap them. For example, in Figure 2.2(b), $T_1$ and $T_2$ could be tiles in a large rectangular grid that can be used to partition both $R$ and $S$. For each partition, PBSM accesses the objects from $R$, the objects from $S$ and performs their join in memory (e.g., using plane-sweep). Since two replicated objects may intersect in multiple tiles (e.g., see $r_4$ and $s_2$ in Figure 2.2(b)), duplicate results may be produced. In order to avoid duplicates, a join result is reported by a tile only a pre-specified reference point (e.g., the top-left corner) of the intersection region is in the tile [21]. Other MASJ approaches include *Spatial Hash Join* [103] and *Scalable Sweeping-Based Spatial Join* [81].

More recent spatial join algorithms consider the potential differences between the joined datasets in the distribution and density. Motivated by a neuroscience application, which requires joining datasets of contrasting density, Pavlovic et al. [106] design a spatial join algorithm that partitions the dense dataset and 'crawls' through the partitions guided by the object locations in the sparse dataset, skipping partitions that do produce any results. Based on the same motivation, a more sophisticated approach was proposed in [107], which adapts the type of partitioning (MASJ or SAMJ) and the join technique used locally, depending on differences in the densities of the two inputs.

#### 2.2.3.3 In-Memory Evaluation

Even with a large main memory that can accommodate the data, plane sweep can be too expensive if directly applied. The main reason behind this is that on a large map containing relatively small rectangles, the chances that two rectangles with intersecting $x$-projections also intersect in the $y$-dimension are low. Hence, plane sweep finds too many candidate pairs that $x$-intersect but do not materialize to actual results.

As a result, in-memory join approaches also consider data partitioning or indexing to accelerate processing. For example, as in PBSM, a grid can be used to break the problem into numerous small instances that can be solved fast. Algorithm *TOUCH* [108] is an effort in this direction, designed for scientific applications that join huge datasets that have different density and skew. TOUCH first bulk-loads an R-tree for one of the inputs using the STR technique [109]. Then, all objects from the second

input are assigned to buckets corresponding to the non-leaf nodes of the tree. Each object is hashed to the lowest tree node, whose MBR overlaps it, but no other nodes at the same tree level do. Finally, each bucket is joined with the subtree rooted at the corresponding node with the help of a dynamically created grid data structure for the subtree. A recent comparison of spatial join algorithms for in-memory data [13] shows that PBSM and TOUCH perform best and that the join cost depends on the data density and distribution. Tauheed et al. [110] suggest an analytical model for configuring the grid of PBSM-like join processing in main memory; however, this model (i) assumes a nested loops evaluation of each partition-partition join and (ii) does not consider using the duplicate avoidance approach of [21].

### 2.2.4 Parallel and Distributed Data Management

Early efforts in parallel and distributed spatial query evaluation have mainly focused on spatial joins, which are more expensive than range queries and they can benefit more from parallelism. The R-tree join (RJ) algorithm [18] and PBSM [14] were parallelized in [111] and [112, 113], respectively.

With the advent of Hadoop, research on spatial data management has shifted to developing distributed systems for spatial data [114, 16, 17, 115, 23, 24]. Spatial data in *Hadoop-GIS* [16] are partitioned using a hierarchical grid, wherein high density tiles are split to smaller ones, in order to handle data skew. The nodes of the cluster share a *global tile index* which can be used to find the HDFS files where the contents of the tiles are stored. For query evaluation, an implicit parallelization approach is followed, which leverages MapReduce. That is, the partitioned objects are given IDs based on the tiles they reside and finding the objects in each tile can be done by a map operation. Spatial queries are implemented as MapReduce workloads. In the *SpatialHadoop* [17], data are also spatially partitioned, but offers different options for partitioning, based on different spatial indices (i.e., grid based, R-tree based, quad-tree based, etc.). The Master node holds a global spatial index for the MBRs of each of the HDFS file blocks. A local index is built at each physical partition and used by map tasks.

Spark-based implementations of spatial data management systems [115, 23, 24] apply similar partitioning approaches. The main difference to Hadoop-based implementations is that data, indices, and intermediate results are shared in the memories

of all nodes in the cluster as *resilient distributed datasets* (RDDs) and can be made persistent on disk. Unlike SpatialSpark [115] and GeoSpark [24] which are built on top of Spark, Simba [23] has its own native query engine and query optimizer, however, Simba does not support non-point geometries. Pandey et al. [20] conduct a comparison between in-memory spatial analytics systems and find that they scale well in general, although each one has its own limitations. Similar conclusions are drawn in another study [116].

Distributed spatial data management systems focus on data partitioning and not on query evaluation at each partition. In other words, emphasis is given on scaling out (i.e., making the cost anti-proportional to the number of nodes), rather than on per-node scalability (i.e., reducing the computational cost per node) and multi-core parallelism.

## 2.3 Tree-like Structures for Relational Data

### 2.3.1 B-tree

The $B^+$-tree is considered the de-facto access method for relational data, having substantial advantages over hash-based indexing with respect to construction cost, support of range queries, sorted data access, concurrency control, etc. [117, 118]. It was firstly introduced as B-tree [119, 120] and it was disk based index. Their main difference was that $B^+$-tree have keys only in the leafs nodes, while B-tree can have keys in both inners and leafs nodes. Then, red black trees [121, 122] are introduced, which are self-balancing binary search trees in main memory, designed to maintain efficient in search insertions and delete operations. O'Neil et al. [123] proposed Log-Structured Merge-tree (LSM-tree), a disk-based data structure that effectively supports a vast number of inserts and deletes over an extended period. To achieve this, employs an algorithm that postpones and groups index updates, efficiently cascading these changes from an in-memory component through one or more disk components, in a way similar to the merge sort process. To support range queries efficiently, the B-tree evolved to a $B^+$-tree, where all keys appear sorted in the (linked) leaf nodes and some keys are replicated in the non-leaf nodes acting as domain separators.

As memory sizes grow, the interest has shifted to in-memory access methods [124]. A set of rebalancing operations leading to significantly more efficient updates

for red black trees was proposed in [36]. One of the first in-memory indices was the T-tree [37], which combines the intrinsics of binary search trees with the storage characteristics of B-trees and it is very effective when the tree and the data are keept in main memory. Rao and Ross [38] were the first to consider the impact of cache misses in memory-based data structures; they proposed *Cache-Sensitive Search Trees* (CSS-trees), in which every node has the same size as the cache-line of the machine and does not need to keep pointers for the links between nodes, but offsets that can be calculated by arithmentic operations. Based on previous technique, Rao and Ross [39] also proposed the Cache Sensitive $B^+$-tree ($CSB^+$tree), which achieves cache performance close to CSS-Trees, while having the advantages of a $B^+$-tree. Chen et al. [125] highlighted how prefetching can significantly improve the performance of index structures by reducing memory access latency. The same authors later proposed a fractal prefetching technique that bulk-reads $B^+$-tree nodes in a hierarchical manner, minimizing both cache and disk accesses [126]. Hankins et al. [127] proved that the optimal index performance in a $CSB^+$tree, can be achieved by balancing the cache misses, instruction count and TLB misses; they include an extended analysis of how the size of the node affects performance. PkT-trees and pkB-trees [40] are in-memory variants of the T-tree and the B-tree, respectively, that use partial-keys (fixed-size parts of keys), which slightly increase the space overhead, but reduce cache misses and improve search performance. Zhou and Ross [128] investigated buffering techniques, based on fixed-size or variable-sized buffers, for memory index structures, aiming to avoid cache trashing and to improve the performance of bulk lookup in relation to a sequence of single lookups. Graefe and Larson [129] provided a survey of all the available techniques that can improve the perfomance of $B^+$-tree by exploting CPU caches. Interpolation search techniques on a $B^+$-tree were studied in [130].

### 2.3.2 (Data) parallelism in B-trees

Modern CPUs, where multiple comparisons can be performed by a single SIMD instruction and the evolution of GPUs opened new perspectives for in-memory index structures. In an early work, Zhou and Ross [131] explored how SIMD instructions can be used to optimize key database operations, like scans, joins, and filtering. The inherent parallelism of SIMD and the avoidance of branch misprediction can greatly improve the performance of an index. Schlegel et al. [67] present two k-ary search

algorithms (find which partition of sorted data out of k contains a search key), one for sorted arrays and one using linearized k-ary search trees, using SIMD instructions. FAST [41], designed for modern CPUs and GPUs, optimizes k-ary tree search by leveraging architecture-specific features like cache locality, SIMD parallelism on CPUs, and massive parallelism on GPUs. It organizes binary sub-trees in the memory to decrease cache misses and memory latency. [42] introduced a "braided" $B^+$-tree structure optimized for parallel searches on GPUs, enabling lock-free traversal using additional pointers. By leveraging CUDA for parallelism and optimizing memory access, the approach significantly improves search performance over traditional CPU-based methods, especially for large datasets. Kaczmarski [44] proposed the GPU $B^+$-tree, a bottom-up $B^+$-tree construction and maintenance technique using CPU and GPU for bulk-loading and updates. Bw-Tree [43] is a highly scalable and latch-free $B^+$-tree variant optimized for modern hardware platforms, including multi-core processors and flash storage. Bw-Tree uses a mapping table for indirection and delta updates for efficient modifications. It also does not use locks, so it can achieve high throughput and is particularly suited for workloads, that require high concurrency and efficient write handling. Hybrid $B^+$-tree [45] leverages both CPU and GPU resources to optimize in-memory indexing on heterogeneous computing platforms. It dynamically balances the workload between the CPU and GPU and exploits high GPU parallelization. Hybrid $B^+$-tree delivers improved performance for search and indexing tasks, particularly in environments with high concurrency and large data volumes. Yan et al. [46] proposed a $B^+$-tree tailored for GPU and SIMD architectures. This structure decouples the "key region", which contains keys of the $B^+$-tree with the "child region", which is organized as a prefix-sum array and stores only each node's first child index in the key region. They also provided two optimizations: first, they partially sort queries to enable coalesced memory access, and second, they group queries to decrease unnecessary comparisons within a warp, thereby reducing warp execution time. Kwon et al. [47] recently proposed $DB^+$-tree, a $B^+$-tree with partial keys, that utilizes SIMD and other sequential instructions for fast branching. PALM [132] is a parallel latch-free variant of the $B^+$-tree, that is optimized for multi-core processors, enabling concurrent search and update operations. Several other papers explore the implementation of B-trees on newer hardware platforms, including flash memory, [133, 134, 135, 136, 137, 138, 139, 140], Non-Volatile Memory [141, 142] and Hardware Transactional Memory [143].

### 2.3.3 Other in-memory access methods

Besides B-trees, other data structures have also been used for in-memory indexing, especially trie-based ones. The trie [144, 48, 145] was originally used for storing and searching strings, aiming at path compression. The internal node representation in the first generation of tries uses lists [146] and arrays [144]. Morrison [48] was the first that introduced path compression and proposed Patricia algorithm, which optimizes the trie data structure by compressing long paths, making it more memory-efficient and faster for certain types of searches. HAT-trie [147, 49] is an in-memory, cache-conscious data structure designed for efficient string storage and retrieval. It combines aspects of both tries and hashing to optimize performance, particularly in terms of memory access patterns. Aktipis and Zobel [148] proposed new update techniques for string processing, which are implemented aiming for disk-based applications. The generalized prefix tree (trie) [50] is an in-memory index structure for arbitrary data types, which uses a variable prefix length. This method attempts to overcome the common problems of tries, such as large trie height and memory requirements. To address these flaws, they introduced four techniques,bypass jumper array (the core concept of the technique bypass jumper array is to bypass trie nodes for leading zeros of a key), trie expansion (tuples can be at any level rather than only in leaves), memory preallocation, and reduced pointers (store only the offset within the preallocated memory instead of the pointer to a certain memory position itself). Kissinger at al. [51] proposed KISS-TREE, a latch-free in-memory index, based on the generalized prefix tree [50], which uses memory management functionalities (like MMU) provided by the operating system and compression mechanisms to minimize the number of memory accesses. The techniques that they introduced to achieve that are direct addressing, on-demand allocation, compact pointers, and compression. Masstree [52] is a persistent data structure that combines aspects of $B^+$-tree and trie. It keeps all the data in memory and shares them with all cores to preserve load balance, maintains high concurrency using optimistic concurrency control, and can support keys with shared prefixes efficiently. The fanout of the tree was chosen to minimize total DRAM delay when descending the tree with prefetching.

Leis et al. [53] proposed a fast and space-efficient in-memory trie indexing structure using modern hardware, called ART. ART dynamically adjusts its node sizes (4, 16, 48, or 256 bytes), based on the number of children, providing a compact and

cache-efficient representation. It uses lazy expansion (inner nodes are only created if they are required to distinguish at least two leaf nodes) and path compression (removes all inner nodes that have only a single child) to improve space utilization and search performance. Leis et al. proposed two synchronization protocols for ART in [149], which have good scalability despite relying on locks: optimistic lock coupling and the read-optimized write exclusion (ROWEX) protocol. Height Optimized Trie (HOT) [55] is an in-memory trie-based index that reduces tree height through path compression and node merging. It combines several nodes from a binary Patricia trie [62] into compound nodes with a maximum fanout, to reduce the height of the structure. Additionally, it uses partial keys for each original key, determined by the bit divisions at each node, to conserve memory. Each node's layout is designed for efficiency, ensuring compactness and enabling fast searches with the use of SIMD instructions. Zhang et al. [54] presented a hybrid index, that uses two different data structures, aiming to achieve memory efficiency and high-performance. Their key idea is that certain data items are accessed more often than others and thus are more likely to be accessed again in the near future. The first structure is dynamic and provides fast insertions and accesses and the second structure is more compact and read-optimized to serve reads for colder data. SuRF (Succinct Range Filter) [56] uses ideas behind the bloom filter to support range queries efficiently. It leverages succinct tries to provide a space-efficient solution for range query filtering.

### 2.3.4 Learned Indexing

The advent of fast and accurate machine learning techniques inspired the design of a new type of index structure, called *learned index* [57]. Learned index uses ML models to "learn" the data distribution and predict the location of data within a dataset aiming to compress the size of the index and reduce the number of memory accesses. The main idea is to learn a cumulative distribution function (CDF) of the keys and define a Recursive Model Index (RMI) [57] that replaces the inner nodes of a $B^+$-tree by a hierarchy of models that can predict very fast the position of the search key. Local search is used to identify the true position of a key if the prediction of RMI has an error. At the top level of RMI, a model provides a prediction for the model to use at the next level, until a model above the sorted key array predicts the position of the search key. This prediction is refined through subsequent levels of models, each

focusing on specific data regions. The lower-level models become more precise until the final level, where a traditional binary search or another simple search mechanism can be used. Galakatos et al [58] propoosed FITing tree, a data-aware index structure, based on RMI, that adjusts its structure based on how the data is spread out. FITing tree maintains a hierarchical structure similar to that of $B^+$-tree, but instead of rigidly dividing data into blocks, it uses the learned interpolation function to predict a key's location within each block. FITing uses linear interpolation models that learn a piecewise linear approximation of the cumulative distribution of the keys, allowing the index to estimate the position of a search key more accurately than fixed, rule-based methods. PGM-index [59] is a fully-dynamic, compressed learned index that uses a piecewise geometric model to efficiently index and query large datasets, Its major contribution is its provable worst-case bounds on query time and space usage. The RadixSpline (RS) [63] learned index can be constructed in a single traversal of sorted data.

ALEX [60] is a learned index structure, based on RMI, designed to efficiently handle updates, using exponential search. It retains a hierarchical structure, like a $B^+$-tree, where inner nodes use linear regression models. ALEX utilizes a *gapped array* layout that gracefully distributes extra space between elements based on the model's predictions, enabling faster insertions and lookups. In comparison, a conventional $B^+$-tree stores all the empty space at the end of the array, which can be less efficient for inserts. CARMI [61] incorporates data partitioning into the construction of RMI and allows for data updates. NFL [62] is a two-stage Normalizing-Flow-Learned index framework that, instead of directly segmenting the CDF curve, initially utilizes the Numerical Normalizing Flow to convert the original keys into nearly uniformly distributed keys, resulting in a CDF curve that is approximately linear. Subsequently, the transformed keys effectively approximate the transformed CDF. LIPP [64] is an updatable learned index that also uses gaps to facilitate updates. One of the major challenges of learned indices is the approximation error when predicting the position of a key. LIPP proposed methods to improve the precision of key predictions, reducing the need for local search after model predictions, leading to faster lookups. The authors introduce a new metric for determining the layout of tree nodes and propose a dynamic adjustment strategy to keep the tree height tightly constrained. DILI [65], is a yet another distribution-driven learned tree for main memory that uses linear regression models for each node to map keys to corresponding children or records.

Learning is done during DILI's bulk loading construction, which is performed in two phases. In the first phase, a balanced bottom-up tree is created using linear regression models that account for both global and local key distributions. In the second phase, DILI is constructed in a top-down approach based on previous constructed bottom-up tree, customizing the fanout of internal nodes based on the local key distributions. Zhang et al. [66] proposed Hyper, an in-memory and multi-threaded learned index, that uses hybrid construction and runtime adjustment techniques to improve query performance and memory footprint. Their core idea, is that leaf nodes need more memory space than inner nodes, but non-leaf nodes are very important, because they should be more accurate. So, they create a hybrid construction, where the leaf nodes are created bottom-up to reduce the overall memory overhead and the inner nodes are conctructed top-down, allowing more memory consumption to achieve better and more accurate predictions. Other learned indices have also been proposed to support concurrency [150, 151, 152, 153], for bloom filters [154], and for persistent memory [155]. [156, 157] and [158] provide comprehensive evaluations on updatable learned indices and traditional indices including many important findings, based on tests on several real-world datasets.

**Algorithm 2.1** Forward Scan based Plane Sweep for Spatial Join

**Input** : collections of rectangles $R$ and $S$

**Output** : set $J$ of all intersecting rectangles $(r, s) \in R \times S$

1: **sort** $R$ and $S$ by lower $x$-endpoint $x_l$
2: $r \leftarrow$ first rectangle in $R$
3: $s \leftarrow$ first rectangle in $S$
4: **while** $R$ and $S$ not depleted **do**
5:  **if** $r.x_l < s.x_l$ **then**
6:  $s' \leftarrow s$
7:  **while** $s' \neq$ null ***and*** $r.x_u \geq s'.x_l$ **do**
8:  **if** $r.y$ intersects $s'.y$ **then**
9:  **output** $(r, s')$; ▷ update result
10:  **end if**
11:  $s' \leftarrow$ next rectangle in $S$; ▷ scan forward
12:  **end while**
13:  $r \leftarrow$ next rectangle in $R$
14:  **else**
15:  $r' \leftarrow r$
16:  **while** $r' \neq$ null ***and*** $s.x_u \geq r'.x_l$ **do**
17:  **if** $r'.y$ intersects $s.y$ **then**
18:  **output** $(r', s)$; ▷ update result
19:  **end if**
20:  $r' \leftarrow$ next rectangle in $R$; ▷ scan forward
21:  **end while**
22:  $s \leftarrow$ next rectangle in $S$
23:  **end if**
24: **end while**

# Chapter 3

# Parallel Strategies For In-Memory Partitioning On Interval Joins



In this chapter, we discuss our study on the problem of interval joins. Our focus is on implementing parallel strategies that can be used in both domain-based and hash-based partitioning to improve the performance of the partitioning phase. We do not go into detail about the join algorithm for each partition but briefly mention a traditional join algorithm that we use.

**Outline** In Section 3.1, we explain domain-based partitioning. Strategies for parallel partitioning are introduced in Section 3.2 and the plane sweep algorithm is presented in Section 3.3. Section 3.4 presents our experimental results, and Section 3.5 provides the conclusion.

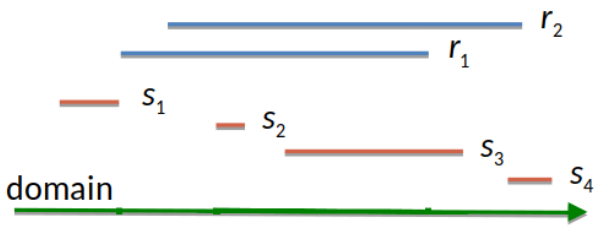



Figure 3.1: Intervals Example.

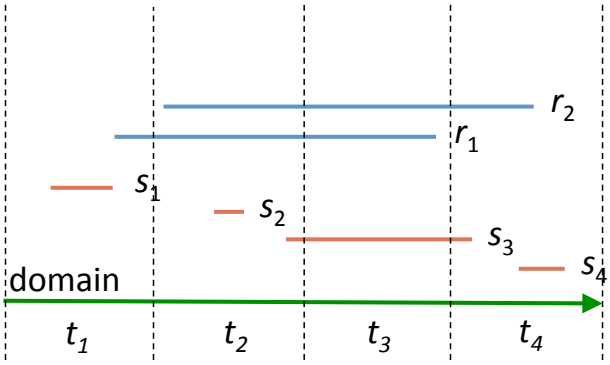



Figure 3.2: Domain-based partitioning of the intervals in Figure 3.1; the case of 4 domain stripes $t_1 \dots t_4$.

## 3.1 Domain-based Partitioning

Domain-based partitioning algorithm [159, 87, 85] for parallel interval joins (Algorithm 3.1) involves two phases. The first phase (Lines 1–17) splits the domain uniformly into $k$ equally-sized and non-overlapping stripes; a partition $R_j$ (resp. $S_j$) is created for each domain stripe $t_j$. Let $t_{start}$, $t_{end}$ denote the stripes that cover $r.start$, $r.end$ of an interval $r \in R$, respectively. Interval $r$ is first assigned to partition $R_{start}$ created for stripe $t_{start}$. Then, $r$ is *replicated* across stripes $t_{start+1}...t_{end}$. During the second phase (Lines 18–20), the domain-based paradigm computes $R_j \bowtie S_j$ for every domain stripe $t_j$, independently. To avoid producing duplicate results, a join result $(r, s)$ is reported if *at least one* of the involved intervals is not a replica. We can easily prove that if for both $r$ and $s$ the start endpoint is not in $t_j$, then $r$ and $s$ should also intersect in the previous stripe $t_{j-1}$, therefore $(r, s)$ will be reported by another partition-join.

We assume that we are allocating 4 CPU threads for computing $R \bowtie S$. To fully take advantage of parallelism, we assign each partition-join to a separate thread. The domain-based paradigm will first split the domain into the 4 disjoint stripes pictured

**Algorithm 3.1** Domain-based Partitioning

**Input**: collections of intervals $R$ and $S$, number of partitions $k$

**Output** : all intersecting interval pairs $(r, s) \in R \times S$

1: **split** domain into $k$ stripes
2: **for all** interval $r \in R$ **do** ▷ partition $R$
3: $t_{\mathsf{start}} \leftarrow$ domain stripe covering $r.\mathsf{start}$
4: $t_{\mathsf{end}} \leftarrow$ domain stripe covering $r.\mathsf{end}$
5: **add** $r$ to partition $R_{\mathsf{start}}$
6: **for all** stripe $t_j$ inside $(t_{\mathsf{start}}, t_{\mathsf{end}}]$ **do**
7: **replicate** $r$ to partition $R_j$
8: **end for**
9: **end for**
10: **for all** interval $s \in S$ **do** ▷ partition $S$
11: $t_{\mathsf{start}} \leftarrow$ domain stripe covering $s.\mathsf{start}$
12: $t_{\mathsf{end}} \leftarrow$ domain stripe covering $s.\mathsf{end}$
13: **add** $s$ to partition $S_{\mathsf{start}}$
14: **for all** stripe $t_j$ inside $(t_{\mathsf{start}}, t_{\mathsf{end}}]$ **do**
15: **replicate** $s$ to partition $S_j$
16: **end for**
17: **end for**
18: **for all** domain stripe $t_j$ **do**
19: **compute** $R_j \bowtie S_j$ ▷ FS and variants
20: **end for**

in Figure 3.2, and then assign and replicate (if needed) the intervals into 4 partitions for each collection; $R_1 = \{r_1\}$, $R_2 = \{\hat{r}_1, r_2\}$, $R_3 = \{\hat{r}_1, \hat{r}_2\}$, $R_4 = \{\hat{r}_1\}$ for $R$ and $S_1 = \{s_1\}$, $S_2 = \{s_2, s_3\}$, $S_3 = \{\hat{s}_3\}$, $S_4 = \{\hat{s}_3, s_4, s_5\}$ for $S$, where $\hat{r}_j$ (resp. $\hat{s}_j$) denotes the replica of an interval $r_i \in R$ (resp. $s_i \in S$) inside stripe $t_j$. Last, the paradigm will compute partition-joins $R_1 \bowtie S_1$, $R_2 \bowtie S_2$, $R_3 \bowtie S_3$ and $R_4 \bowtie S_4$. Note that $R_3 \bowtie S_3$ will produce no results because all contents of $R_3$ and $S_3$ are replicas, while $R_4 \bowtie S_4$ will only produce $(r_1, s_4)$ but not $(r_1, s_3)$ which will be found in $R_2 \bowtie S_2$.

Also, as opposed to previous work that also applies domain-based partitioning (e.g., [87, 85]), this technique avoids the production and elimination of duplicate join results.

## 3.2 Strategies for Parallel Partitioning

We next elaborate on how the partitioning process can benefit from modern parallel hardware. We discuss three strategies applicable on the domain-based partitioning; in the next section, we carefully evaluate these strategies for each partitioning type. As a common feature, all strategies operate in three phases. During the first phase, all available CPU cores or threads are employed to calculate the cardinality of each $|R_j|$ and $|S_j|$ partition. During the second phase, the threads are employed to allocate the space required to store every partition in main memory and then physically partition the input collections. Finally, again all available threads are used to sort and index (if needed) the input partitions, depending on the interval join algorithm to be used.[1] In the following, we detail the first two phases for each partitioning strategy.

One2One. The first strategy was used in [80] for hash-based partitioning but can be straightforwardly applied for the domain-based as well. The idea is to exclusively assign every $R_j$ (resp. $S_j$) partition to a single thread.[2] Under this, the thread executes all phases of the partitioning process for $R_j$. As every partition of the collection is assigned to exactly one thread, the entire partitioning process is essentially divided into smaller independent tasks which run in parallel without the need of synchronization. Strategy 3.2 illustrates a high-level pseudo-code of One2One. After initiating $c$ parallel threads in Line 1, every thread executes the first and the second phase of the partitioning independently in Lines 3–8. Consider thread $j$. During the first phase in Lines 3–5, thread $j$ is assigned $\frac{k}{c}$ partitions for the input collection $R$, where $k$ is the number of requested partitions and $c$ is the number of available threads. Specifically, the thread gets all partitions in the range from $\left((j-1)\cdot\frac{k}{c}+1\right)$ to $\left(j\cdot\frac{k}{c}\right)$ Then, it scans collection $R$ to count how many intervals will be contained inside its assigned partitions. Last, during the second phase in Lines 6–8, every thread allocates the space needed to store their assigned partitions and then, scans for the second time the input collection to fill these partitions.

Despite its simplicity, the One2One strategy has two important drawbacks. First, it requires multiple scans over the input; to be precise, the collection is scanned $2\cdot c$ times. Second, the strategy cannot cope with skewed data distributions; essentially,

[1] Recall that every partition may take part in multiple joining tasks. Hence, we choose to introduce a separate sorting/indexing phase instead of having this step integrated inside the join algorithm.

[2] In general, the number of partitions per input may exceed the number of available threads in which case, every thread is responsible for multiple partitions.

**Algorithm 3.2** One2One

**Input** : collection of intervals $R$, number of partitions $k$, number of threads $c$

**Output** : partitions $\{R_1, \ldots, R_k\}$

**Variables**: counters $\{|R_1|, \ldots, |R_k|\}$

1: **create** $c$ parallel threads
2: **for all** thread $j$ **do** ▷ executed in parallel
3: **assign** the $j$-th set of $\frac{k}{c}$ partitions to the thread
4: **read** intervals from $R$
5: **calculate** counters $\{|R_{\left((j-1)\frac{k}{c}+1\right)}|, \ldots, |R_{\left(j\cdot\frac{k}{c}\right)}|\}$
6: **allocate** memory space for assigned partitions
7: **read** intervals from $R$
8: **fill** partitions $\{R_{\left((j-1)\cdot\frac{k}{c}+1\right)}, \ldots, R_{\left(j\cdot\frac{k}{c}\right)}\}$
9: **end for**
10: **return** $\{R_1, \ldots, R_k\}$

the cost of the entire partitioning process is dominated by the cost of processing the largest partition. In what follows, we discuss two partitioning strategies that address these issues.

Temps. The key idea for fast partitioning is to assign parts of the input collection to the available threads instead of entire partitions. Under this, every thread reads a chunk from the input containing $\frac{|R|}{c}$ intervals, and builds a temporary local partitioning. The input chunks should be disjoint such that the parallel threads operate completely independently. Every thread performs a first scan of its assigned intervals to count how large its local partitions will be, then allocates the required space in main memory and reads again the intervals to fill the partitions. Finally, after all threads have finished, the local partitionings are unified into the final result as the last step.

Strategy 3.3 illustrates a high-level pseudo-code of Temps. In Lines 2–7, every thread scans (two times) its assigned chunk of the input collection to create a local partitioning. Specifically, thread $j$ gets the $j$-th chunk of $\frac{|R|}{c}$ input intervals and produces local partitioning $\{R_1^j, \ldots, R_k^j\}$; notice that local partitionings contain the same number of partitions as the final result. To count the cardinality of its local partitions, the thread maintains private local counters $\{|R_1^j|, \ldots, |R_k^j|\}$. After all local

**Algorithm 3.3** Temps

**Input** : collection of intervals $R$, number of partitions $k$, number of threads $c$

**Output** : partitions $\{R_1, \ldots, R_k\}$

**Variables:** global counters $\{|R_1|, \ldots, |R_k|\}$, local partitions $\{R_1^j, \ldots, R_k^j\}$ and local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for every parallel thread $j$

1: **create** $c$ parallel threads
2: **for all** thread $j$ **do** ▷ executed in parallel
3: **read** the $j$-th chunk of $\frac{|R|}{c}$ intervals from $R$
4: **calculate** local counters $\{|R_1^j|, \ldots, |R_k^j|\}$
5: **allocate** memory space for $\{R_1^j, \ldots, R_k^j\}$
6: **read** the $j$-th chunk of $\frac{|R|}{c}$ intervals from $R$
7: **fill** local partitions $\{R_1^j, \ldots, R_k^j\}$
8: **end for**
9: **wait** until all threads finished ▷ synchronization
10: **for all** partition $R_i$ **do** ▷ executed in parallel
11: **calculate** global counter $|R_i| = \sum_{j=1}^{c} |R_i^j|$
12: **allocate** memory space
13: $R_i \leftarrow \bigcup_{j=1}^{c} R_i^j$ ▷ unify local partitions
14: **end for**
15: **return** $\{R_1, \ldots, R_k\}$

partitionings are built (synchronization barrier in Line 9), Temps unifies them by copying local partitions to a contiguous space allocated in main memory for the final partitions, in Lines 10–14. The domain-based partitioning assign every interval to exactly one local partition; the same holds for the replicas in case of domain-based. Under this, the cardinality for each final partition $R_i$ is calculated as $|R_i| = \sum_{j=1}^{c} |R_i^j|$ and the partition is defined as $R_i^1 \bigcup \ldots \bigcup R_i^c$, where $c$ is the total number of parallel threads and local partitionings. Last, to accelerate this unification step, the Temps strategy assigns the computation of every partition $R_i$ to the next available thread in a round robin fashion.

Compared to One2One, the Temps strategy scans the entire input collection $R$ only twice as every thread now operates on a different chunk of $R$. In addition, as $R$'s chunks are equi-sized, i.e., all contain at most $\frac{|R|}{c}$ intervals, the partitioning load is better distributed to the available threads. But, Temps still exhibits important

**Algorithm 3.4** Divs

**Input** : collection of intervals $R$, number of partitions $k$, number of threads $c$

**Output** : partitions $\{R_1, \ldots, R_k\}$

**Variables**: global counters $\{|R_1|, \ldots, |R_k|\}$, and local counters $\{R_1^j, \ldots, R_k^j\}$ and local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for every parallel thread $j$

1: **create** $c$ parallel threads
2: **for all** thread $j$ **do** ▷ executed in parallel
3: **read** the $j$-th chunk of $\frac{|R|}{c}$ intervals from $R$
4: **calculate** local counters $\{|R_1^j|, \ldots, |R_k^j|\}$
5: **end for**
6: **wait** until all threads finished ▷ synchronization
7: **for all** partition $R_i$ **do** ▷ executed in parallel
8: **calculate** global counter $|R_i| = \sum_{j=1}^{c} |R_i^j|$
9: **allocate** memory space
10: **divide** partition into $c$ logical parts
11: **end for**
12: **wait** until all threads finished ▷ synchronization
13: **for all** thread $j$ **do** ▷ executed in parallel
14: **read** the $j$-th chunk of $\frac{|R|}{c}$ intervals from $R$
15: **fill** $j$-th part of each partition in $\{R_1, \ldots, R_k\}$
16: **end for**
17: **return** $\{R_1, \ldots, R_k\}$

shortcomings. First, for every partition $R_i$, the strategy allocates twice the required space in main memory, i.e., to store both its corresponding local partitions and $R_i$ itself. Second, the strategy introduces an extra costly step, i.e., the unification of local partitioning. Also, the cost of this last step is dominated by the largest partition which is again computed by a single thread.

Divs. To address these shortcomings, we next discuss our last strategy. Strategy Divs shares the same key idea to Temps, i.e., every thread $j$ processes independently the $j$-th chunk of $\frac{|R|}{c}$ input intervals. But, instead of building a temporary local partitioning, the thread directly updates the final partitions. For this purpose, the strategy logically divides every final partition $R_i$ into $c$ parts, i.e., one for each available thread. The extent of each $R_i^j$ part is determined by local counters $|R_i^j|$, which are computed similar

to strategy Temps. With this division, each thread independently fills a dedicated part of $R_i$'s data structure in memory without the need of locking or any type of synchronization.

Strategy 3.4 illustrates a high-level pseudo-code of Divs. Lines 2 and 3 are identical to Strategy 3.3, i.e., a first scan of the input collection determines local counters $\{|R_1^j|, \ldots, |R_k^j|\}$ for each thread $j$. After local counters are computed (synchronization barrier in Line 6), Divs allocates the necessary space in main memory to build every $R_i$ partition (Lines 8–9) and also, logically divides $R_i$ into $c$ parts using its local counters (Line 10). Finally after this preparation step is finished for all partitions (synchronization barrier in Line 12), every thread scans for the second time its assigned input intervals and fills its dedicated part inside the data structure of every partition, in Lines 13–16.

Compared to Temps, the Divs strategy does not allocate extra space for every partition; at the same time, the costly unification step of Temps is entirely avoided. In addition, the largest partition which could become the bottleneck for both strategies One2One and Temps is now filled by multiple threads in parallel achieving a better load balancing.

## 3.3 Plane Sweep for Interval Joins

This section presents the plane sweep algorithm, which is used at each partition-to-partition join. In [18], Brinkhoff et al. presented a different implementation of plane sweep, which performs a *forward scan* directly on the input collections and hence, (i) there is no need to keep track of active sets in a special data structure and (ii) data scans are conducted sequentially. [3] Algorithm 3.5 illustrates the pseudo-code of this method, denoted by FS. First, both inputs are sorted by the start endpoint of each interval. Then, FS *sweeps* a line, which stops at the start endpoint of all intervals of $R$, $S$ in order. For each position of the sweep line, corresponding to the start of an interval, say $r \in R$, the algorithm produces join results by combining $r$ with all intervals from the opposite collection, that start (i) after the sweep line and (ii) before $r.end$, i.e., all $s' \in S$ with $r.start \leq s'.start \leq r.end$ (internal while-loops on Lines 7–11 and 14–18). Excluding the cost of sorting $R$ and $S$, FS conducts $|R|+|S|+|R \bowtie S|$ point

[3] The algorithm originally targets intersection join of 2D rectangles, but it is straightforward to apply for interval joins.

**Algorithm 3.5** Forward Scan based Plane Sweep for Interval Join (FS)

**Input** : collections of intervals $R$ and $S$

**Output** : all intersecting pairs $(r, s) \in R \times S$

```
1:  sort R and S by start endpoint
2:  r ← first interval in R
3:  s ← first interval in S
4:  while R and S not depleted do
5:    if r.start < s.start then
6:      s' ← s
7:      while s' ≠ null and r.end ≥ s'.start do
8:        output (r, s')                          ▷ update result
9:        s' ← next interval in S                 ▷ scan forward
10:     end while
11:     r ← next interval in R
12:   else
13:     r' ← r
14:     while r' ≠ null and s.end ≥ r'.start do
15:       output (r', s)                          ▷ update result
16:       r' ← next interval in R                 ▷ scan forward
17:     end while
18:     s ← next interval in S
19:   end if
20: end while
```

comparisons, in total. Specifically, each interval $r \in R$ (the case for $S$ is symmetric) is compared to just one $s' \in S$ which does not intersect $r$ in the loop at Lines 8–10.

## 3.4 Experiments

Last, we present the second part of our experimental evaluation, which focuses on the parallel computation of interval joins.

Table 3.1: Characteristics of experimental datasets

| | BOOKS | FLIGHTS | GREEND | INFECTIOUS | TAXIS | WEBKIT |
|---|---|---|---|---|---|---|
| Cardinality | 2,312,602 | 445,827 | 110,115,441 | 415,912 | 172,668,003 | 2,347,346 |
| Domain duration (secs) | 31,507,200 | 2,750,280 | 283,356,410 | 6,946,360 | 31,768,287 | 461,829,284 |
| Distinct endpoints | 5,330 | 41,975 | 182,028,123 | 81,514 | 29,873,023 | 174,471 |
| Shortest interval (secs) | 1 | 1,260 | 1 | 20 | 1 | 1 |
| Avg. interval duration (secs) | 2,201,320 | 8,790 | 15 | 20 | 758 | 33,206,300 |
| Longest interval (secs) | 31,406,400 | 42,300 | 59,468,008 | 20 | 2,148,385 | 461,815,512 |

### 3.4.1 Setup

Our parallel-threaded analysis was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz, with 20 threads, running CentOS Linux 7.3.1611. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`.

**Datasets**. We experimented with 6 real datasets, the majority of which was used in recent literature on interval joins; Table 3.1 details the characteristics of the datasets. BOOKS [159] records all transactions at Aarhus public libraries in 2013 (https://www.odaa.dk); valid times indicate the periods when a book is lent out. FLIGHTS [83] records domestic flights in USA during January 2016 (https://www.bts.gov); valid times indicate the duration of a flight. GREEND [77, 160] records power usage data from households in Austria and Italy from January 2010 to October 2014; valid times indicate the period of a measurement. INFECTIOUS [77, 161] stores visiting information from the "INFECTIOUS: stay Away!" exhibition at Science Gallery in Dublin, Ireland, from May to July 2009; valid times indicate when a contact between visitors occurred. TAXIS records taxi trips (pick-up, drop-off timestamp) from New York City (https://www1.nyc.gov/site/tlc/index.page) in 2013; valid times indicate the duration of each ride. WEBKIT [159, 83, 76] records the file history in the git repository of the Webkit project from 2001 to 2016 (https://webkit.org); valid times indicate the periods when a file did not change.

### 3.4.2 Partitioning Strategies

Figure 3.3 reports the domain-based partitioning time for strategies One2One, Temps and Divs while varying the number of partitions; for the tests, we set again $|R| = |S|$ and used up to 20 parallel threads. We observe that Divs is the most efficient and

most robust strategy for parallel partitioning; on the largest datasets GREEND and TAXIS, Temps is competitive to Divs but still slower. However, One2One is clearly the slowest strategy in all cases; its time is severely affected by the increase in the number of partitions exhibiting also a "staircase" pattern (more obvious in Figures 3.3(c) and (e)). For domain-based partitioning we also need to replicate an interval to all overlapping stripes; the replication cost naturally increases with the number of partitions. Regarding the "staircase" pattern, notice that One2One's time essentially goes up every 20 partitions. Consider for example the increase from 20 to 40 partitions. At first, every thread builds exactly one partition. When we increase the number of partitions to 21, this extra partition will be assigned as a second task to one of the available threads. The total time of this thread will increase and dominate the overall partitioning time. Adding more partitions will not change this overall time because there still threads assigned one partition unless the total number of partitions grows higher than 40.

## 3.5 Conclusions

We explored the problem of parallel interval joins and proposed three partitioning strategies, applicable to both hash-based and domain-based partitioning, which utilize parallelism to improve join performance. These strategies are One2One, Temps, and Divs, with Divs delivering the best results.

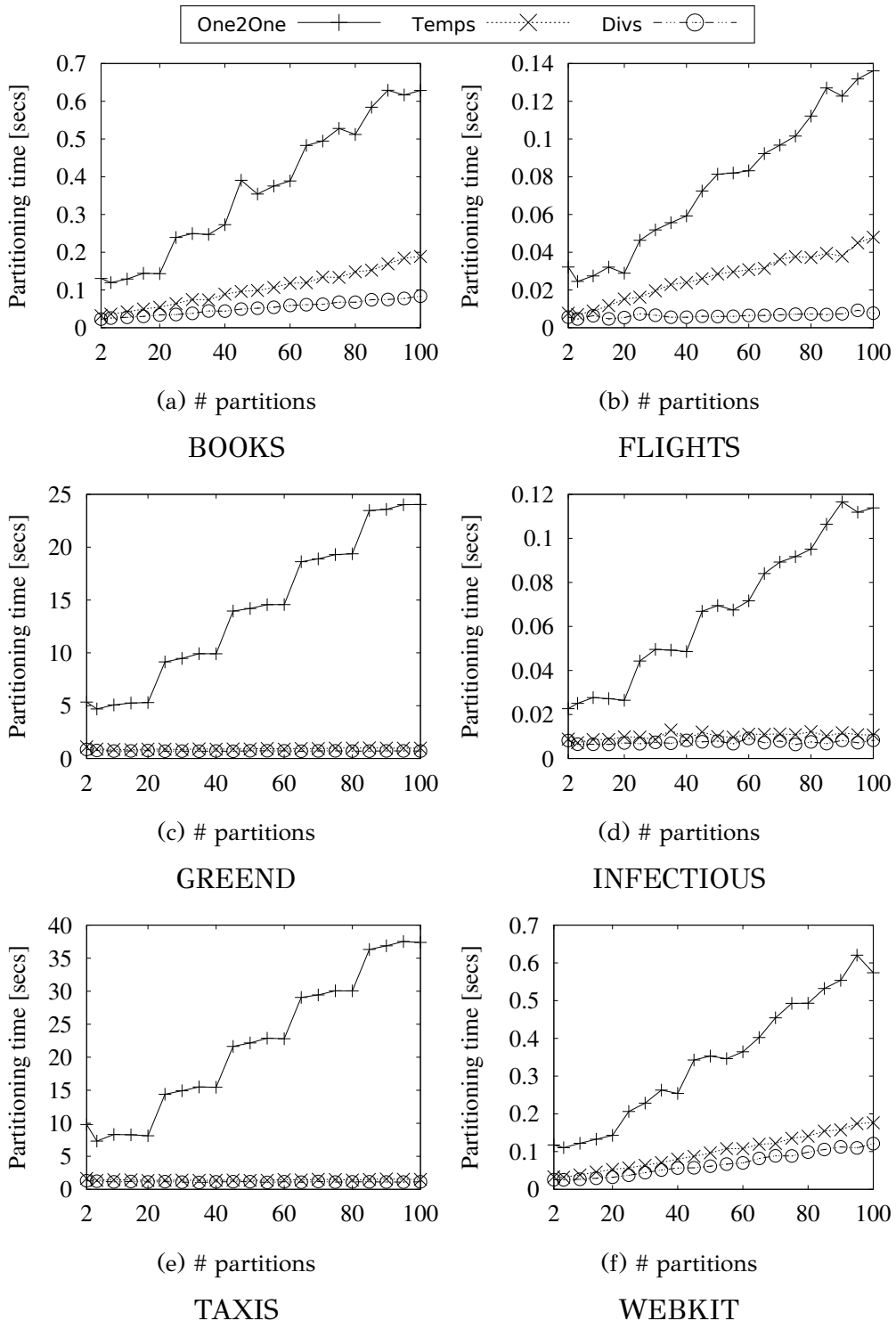



Figure 3.3: Tuning domain-based partitioning: strategies, $|R| = |S|$ and 20 threads.

# Chapter 4

# In-Memory And Parallel Evaluation Of Spatial Queries



Non-point spatial objects (e.g., polygons, linestrings, etc.) are ubiquitous and their effective management is always timely. We study the classic problem of indexing non-point objects, considering the current state of commodity hardware, having relatively large memory and the ability of parallel multi-core processing. In view of this, In Section 4.1 we study the in-memory and parallel evaluation of spatial joins, by tuning a classic partitioning based algorithm. Our study shows that, compared to a straightforward implementation of the algorithm, performance can be improved significantly. We also show how to select appropriate partitioning parameters based on data statistics, in order to tune the algorithm for the given join inputs. Our parallel implementation scales gracefully with the number of threads reducing the cost of the join to at most one second even for join inputs with tens of millions of rectangles. We also propose (Section 4.2) a secondary partitioning technique for space-oriented partitioning indices (e.g., grids), which improves their performance significantly, by avoiding the generation and elimination of duplicate results. Our approach is novel

and of a high impact, as (i) it is extremely easy to implement and (ii) it can seamlessly be applied as part of any space-partitioning index, (iii) it can directly be used by distributed spatial data management systems. We show how our approach can be used to boost the performance of range queries and spatial intersection joins. We also show how we can avoid performing the expensive refinement step of a range query for the majority of objects and study the efficient processing of numerous queries in batch and in parallel. Extensive experiments on real datasets confirm the superiority of space-oriented partitioning over data-oriented partitioning and the advantage of our approach against alternative duplicate elimination techniques. We also show that our partitioning technique, paired with optimized partition-to-partition join algorithms, typically reduces the cost of spatial joins by around 50%.

**Outline** The rest of the Chapter is organized as follows. Section 4.1.1 presents the PBSM directions along which we tune the performance of the algorithm. Section 4.1.2 presents our parallel implementation of the algorithm. In Section 4.1.3, we provide a detailed experimental evaluation. Section 4.2.1 introduces our secondary partitioning scheme and its application to grid-based spatial indexing. Section 4.2.2 shows how spatial range query evaluation can benefit from our indexing scheme. In Section 4.2.3, we present a filtering condition that applies on the MBRs of the objects and can be used to confirm the inclusion of an object to a range query result, without the need of a refinement step. Section 4.2.4 discusses how numerous range queries that may need to be handled can be processed efficiently and in parallel. In Section 4.2.5, we explain how our secondary partitioning scheme can be applied to spatial joins and its optimizations. An experimental evaluation is presented in Section 4.2.6. Finally, in Section 4.3, we summarize our conclusions.

## 4.1 Parallel In-Memory Evaluation of Spatial Joins

### 4.1.1 Tuning PBSM

As discussed in the introduction (Section 1.2), the most popular spatial join framework follows the multi-assignment, single-join (MASJ) [103] paradigm of PBSM. The reasons behind this can be summarized as follows:

- PBSM assumes no preprocessing or indexing of the data, so it can be applied on dynamically generated spatial data.

- The partitions define independent join tasks that can easily be distributed and/or parallelized.
- The number of join tasks is the same as the number of partitions(as opposed to the number of join tasks of SAMJ approaches which can be much higher).
- Producing duplicate results can be easily avoided.
- Implementing this approach is fairly easy.
- Previous studies [13] have shown that the performance of PBSM can hardly be beaten by more sophisticated approaches based on indexing or adaptive partitioning.

In the following, we explore the directions along which we can tune PBSM to improve its performance. These include determining the number and type of partitions (tiles or stripes), duplicate elimination, and choosing the axis along which we perform plane sweep in each partition. We assume that PBSM uses the plane sweep algorithm of [18] for each partition-partition join.

#### 4.1.1.1 One-dimensional Partitioning

The default partitioning approach for PBSM is a 2D grid, as shown in Figure 1.2a. Still, the same algorithm can be applied if we partition the data space in 1D *stripes*, as shown in Figure 1.2b. The stripes can be horizontal or vertical. Such a partitioning was considered by an *external memory plane sweep* join algorithm [81]; however, the objective of the partitioning there was to define the stripes in a way such that the "horizon" of the sweep line (which runs along the axis of the stripes) fits in memory. Since in this study, we deal with in-memory joins, we do not consider this factor, but we study how the number of partitions affects the cost of the join.

#### 4.1.1.2 Duplicate Elimination

Dittrich and Seeger [21] presented a simple but effective approach for eliminating duplicate results in PBSM. A rectangle pair is reported by a partition-partition join only if the top-left corner of their intersection area is inside the spatial extent of the partition. The pair of intersecting rectangles $(r_1, s_1)$ in Figure 1.2a can be found in both tiles (0,0) and (0,1). However, this result will only be reported in tile (0,0), which contains the top-left corner of the intersection. In other words, the join result will be computed in tile (0,1) but not reported. Hence, for each rectangle pair found to

intersect, a *duplicate* test is performed. Let $[r.x_l, r.x_u]$ and $[r.y_l, r.y_u]$ be the projections of rectangle $r$ on the $x$ and $y$ axis, respectively. Let $[T.x_l, T.x_u]$ and $[T.y_l, T.y_u]$ be the corresponding projections of a tile. The duplicate test for pair $(r, s)$, found to intersect in tile $T$, is the condition:

$$\max\{r.x_l, s.x_l\} \geq T.x_l \wedge \max\{r.y_l, s.y_l\} \geq T.y_l \tag{4.1}$$

**Application to 1D partitioning.** For the 1D partitioning, the duplicate test needs to apply a single comparison (as opposed to the two comparisons of Eq. 4.1). For example, if the stripes are vertical (as in Figure 1.2b), a join result is reported only if $\max\{r.x_l, s.x_l\} \geq T.x_l$.

#### 4.1.1.3 Choosing the Sweeping Axis

When applying plane sweep for a tile (or stripe) $T$, we have to decide along which axis we will sort the rectangles and then sweep them. We devise a model which, given the sets of rectangles $R_T$, $S_T$ inside a tile $T$, determines the sweeping axis to be used. The key idea is to estimate, for each axis, how many candidate pairs of rectangles from $R_T \times S_T$ intersect along this axis. For this purpose, to estimate the number of intersecting projections per axis, we compute histogram statistics. In specific, we subdivide the $x$ and $y$ projections of the tile $T$ into a predefined number of partitions $k$. Then, we count how many rectangles from $R$ and how many from $S$, $x$-intersect each $x$-division of the tile; the procedure for $y$ partitions is symmetric. In this manner, we construct four histograms $H_R^x$, $H_R^y$, $H_S^x$, $H_S^y$ of $k$ buckets each. The number $I_T^x$ of rectangles in $R_T \times S_T$ that $x$-intersect can then be approximated by accumulating the product of the corresponding histogram buckets, i.e.,

$$I_T^x = \sum_{i=0}^{k} \{H_R^x[i] \cdot H_S^x[i]\} \tag{4.2}$$

The smallest of $I_T^x$ and $I_T^y$ determines the chosen sweeping axis (i.e., $x$ or $y$). For large tiles (compared to the size of the rectangles), we set $k = 1000$, while for small tiles $k$ is the number of times the tile's extent is larger than the average rectangle extent. In practice, using all rectangles of $T$ in the histogram construction is too expensive. So, we use a sample of rectangles from $R_T$ and $S_T$ for this purpose. Specifically, for every $\phi$ rectangles that are assigned to tile $T$, we use one for histogram construction. We set $\phi = 100$ by default because it can produce good enough estimates at a low overhead.

**Application to 1D partitioning.** Our model can be straightforwardly applied in case of a 1D partitioning; histogram statistics are now computed for the contents of the vertical or horizontal stripes and the entire domain on the other dimension.

### 4.1.2 Parallel Processing

We parallelize evaluation by splitting its partitioning and joining phases into parallel and independent tasks, while trying to minimize the synchronization requirements between the threads. While the parallel algorithm that we outline here is designed for a single, multi-core machine, it can also be applied (with minor changes) to a cluster of machines. The steps for parallelizing the spatial join to $m$ threads are as follows:

**Partitioning phase**

(1) Determine a division of each input $R$ and $S$ into $m$ equi-sized parts arbitrarily.

(2) Initiate $m$ threads. Thread $i$ reads the $i$-th part of input $R$ and *counts* how many rectangles should be assigned to each of the space partitions (tiles or stripes). Thread $i$ repeats the same process for the $i$-th part of input $S$. Let $|R_T^i|, |S_T^i|$ be the numbers of rectangles counted by thread $i$ for tile $T$ and $R$, $S$, respectively.

(3) Compute $|R_T| = \sum_i^m |R_T^i|$ and $|S_T| = \sum_i^m |S_T^i|$ for each tile $T$. Allocate two memory segments for $|R_T|$ and $|S_T|$ rectangles of each partition $T$.

(4) Initiate $m$ threads. Thread $i$ reads the $i$-th parts of inputs $R$ and $S$ and partitions them. The memory allocated for each of $|R_T|$ and $|S_T|$ is logically divided into $m$ segments based on the $|R_T^i|$'s and $|S_T^i|$'s. Hence, thread 1 will write to the first $|R_T^1|$ positions of $|R_T|$, thread 2 to the next $|R_T^2|$ positions, etc. After all threads complete partitioning, we will have the entire set of rectangles that fall in each tile continuously in memory.

**Joining phase**

(5) Construct two sorting tasks for each tile $T$ (one for $R_T$ and one for $S_T$). Assign the sorting tasks to the $m$ threads.

(6) Construct a join task for each tile $T$ (one for $R_T$ and one for $S_T$). Assign the join tasks to the $m$ threads.

Table 4.1: Datasets used in the experiments

| source | dataset | alias | cardinality | avg. $x$-extent | avg. $y$-extent |
|---|---|---|---|---|---|
| Tiger 2015 | AREAWATER | $T2$ | 2.3M | 0.000007230 | 0.000022958 |
| | EDGES | $T4$ | 70M | 0.000006103 | 0.00001982 |
| | LINEARWATER | $T5$ | 5.8M | 0.000022243 | 0.000073195 |
| | ROADS | $T8$ | 20M | 0.000012538 | 0.000040672 |
| OSM | Buildings | $O3$ | 115M | 0.00000056 | 0.000000782 |
| | Lakes | $O5$ | 8.4M | 0.000021017 | 0.000028236 |
| | Parks | $O6$ | 10M | 0.000016544 | 0.000022294 |
| | Roads | $O9$ | 72M | 0.000010549 | 0.000016281 |

Step 2 is applied in order to make proper memory allocation and prevent expensive dynamic allocations. It also facilitates the output of parallel partitioning for each tile $T$ to be continuous in memory during Step 4. When the model of Section 4.1.1.3 is used, the histograms are computed while loading input data (i.e., in either of Steps 2 and 4).

### 4.1.3 Experimental Analysis

#### 4.1.3.1 Setup

We experimented with Tiger 2015 and OpenStreetMap (OSM) datasets from [17].[1] For each dataset, we computed the MBRs of the objects and came up with a corresponding collection of rectangles. The datasets are normalized so that the coordinates in each dimension take values in $[0, 1]$. Table 4.1 details the datasets we used. The first three datasets are from the collection Tiger 2015 and the last three from the collection OpenStreetMap (OSM). Next to each dataset name we put a short alias indicating its order in the Tiger or OSM collection (i.e., $O3$ means the 3rd dataset from OSM). Dataset cardinality ranges from 2.3M to 115M objects and we tested joins having inputs from the same collection, with similar or various scales. The last two columns of the tables are the relative (over the entire space) average length of the rectangle projections at each axis.

We implemented the spatial join algorithm (all different versions) in C++ and compiled it using `gcc` (v4.8.5). For multi-threading, we used OpenMP. All experiments

[1] http://spatialhadoop.cs.umn.edu/datasets.html

Table 4.2: Sweeping axis effect; queries ordered by runtime

| query | sweeping axis | | adaptive model | |
|---|---|---|---|---|
| | $x$ | $y$ | $I^x$ | $I^y$ |
| $T2 \bowtie T5$ | 8.94s | 16.96s | 8,376 | 19,232 |
| $T2 \bowtie T8$ | 24.52s | 40.72s | 8,895 | 18,660 |
| $O5 \bowtie O6$ | 24.92s | 66.06s | 2,692 | 12,279 |
| $O6 \bowtie O9$ | 216.88s | 444.19s | 3,989 | 11,510 |
| $T4 \bowtie T8$ | 674.50s | 1,360.92s | 8,135 | 19,406 |
| $O9 \bowtie O3$ | 926.14s | 1,681.30s | 4,535 | 11,529 |

were run on a machine with 384 GBs of RAM and a dual 10-core Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz running CentOS Linux 7.3.1611; with hyper-threading, we were able to run up to 40 threads. The reported runtimes include the costs of partitioning both datasets and then joining them.

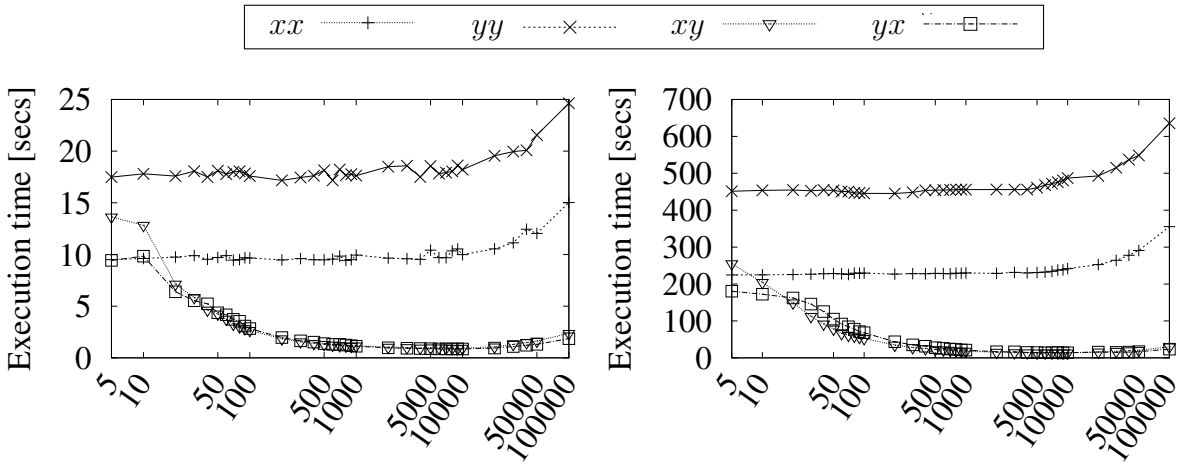
#### 4.1.3.2 Selecting the Sweeping Axis

We first test the effect that the sweeping axis selection (either $x$ or $y$) has on the performance of the algorithm. For this purpose, we chose not to partition the data, but ran the single-threaded plane-sweep join from [18] in the entire dataspace (i.e., modeling the case of a single tile). Table 4.2 reports the execution times per query. We observe that sweeping along the wrong axis may even double the cost of the join. The last column of the table reports the result of running our model (Eq. 4.2). Our model was able to accurately determine the proper sweeping axis in all cases. Note that the cost of this decision-making process is negligible compared to the partitioning and joining cost; even for the largest queries, our model needs less than 10 milliseconds.

#### 4.1.3.3 Evaluation of Partitioning

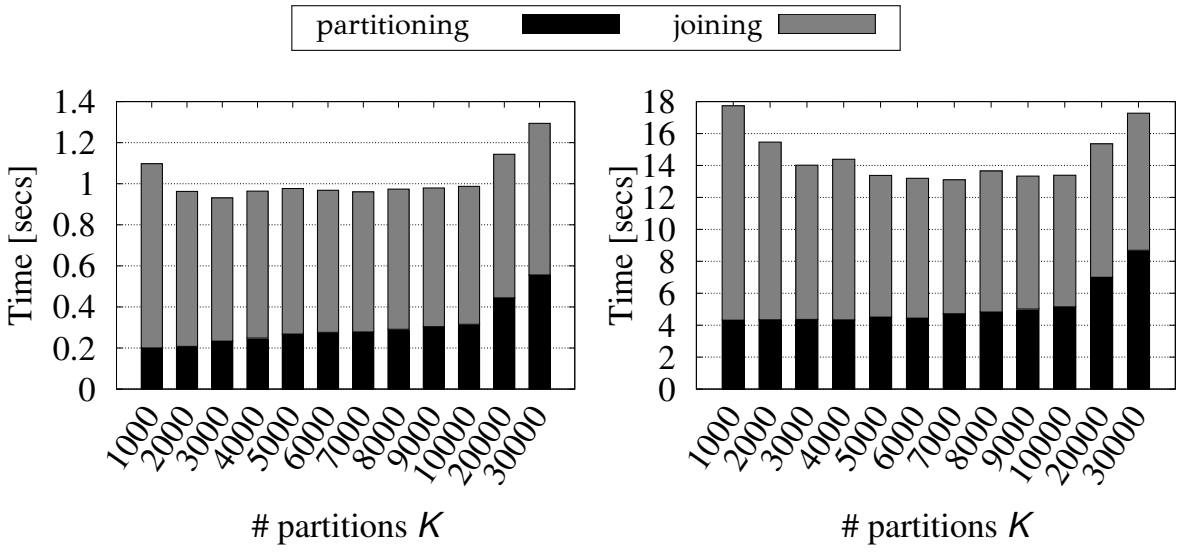
Next, we investigate the impact of partitioning to the performance of the algorithm. We tune 1D and 2D-based PBSM and then compare the two partitioning approaches to each other.

**Tuning 1D Partitioning.** In the next experiment, we evaluate the performance of the algorithm when 1D partitioning is used. Figure 4.1 reports the cost of two spatial join queries while varying the number $K$ of (uniform) 1D partitions. We tested all combinations of partitioning and sweeping axes. For example, $xy$ denotes partitioning




(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 4.1: Tuning 1D partitioning: total execution time

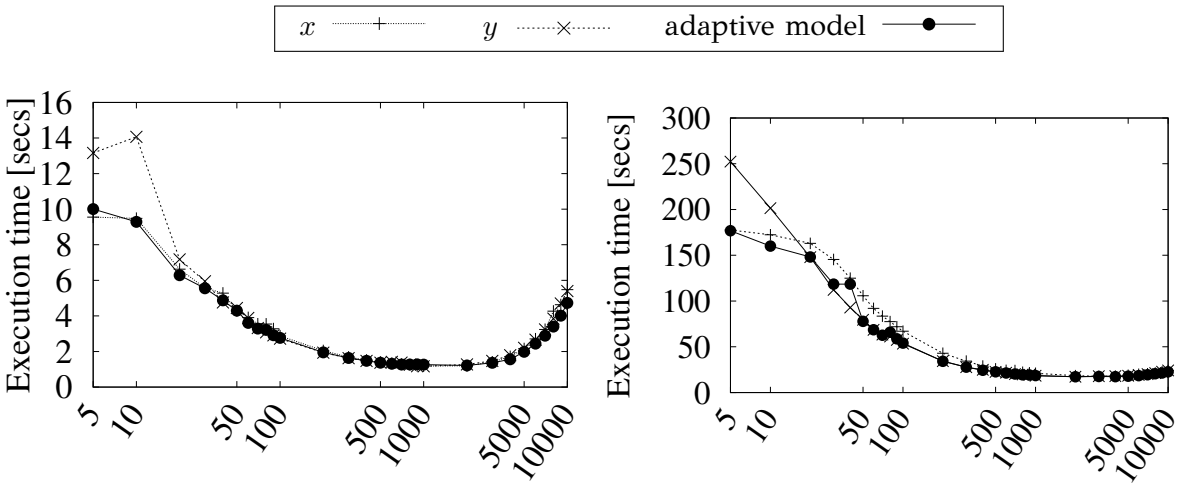



(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 4.2: Tuning 1D partitioning: time breakdown

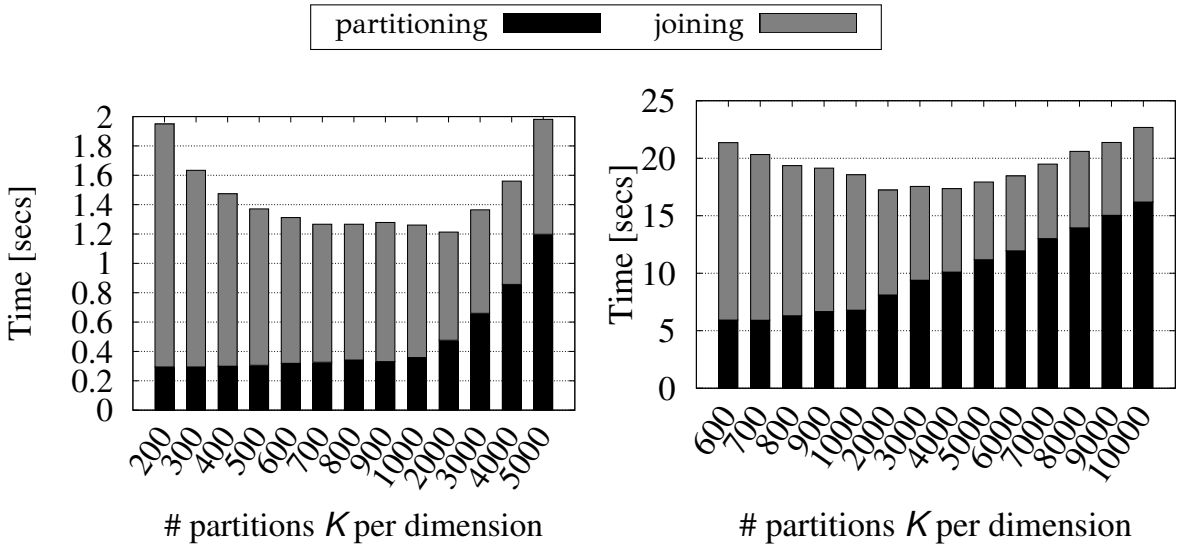
along the $x$ axis (to vertical stripes) and sweeping along the $y$ axis. Note that if the sweeping axis is the same as the partitioning axis (i.e., cases $xx$ and $yy$), the join cost does not drop when we increase the number of partitions $K$. This is expected because, regardless the number of partitions, case $xx$ or $yy$ is equivalent to having no partitions at all and sweeping along the $x$ or $y$ axis in the entire space. When $K$ is too large, the costs of $xx$ and $yy$ increase because the partitions become very narrow and replication becomes excessive. On the other hand, the performance of cases $xy$ and $yx$ improves with $K$ and, after some point, i.e., $K = 2{,}000$, they converge to the same (very low) cost. The costs of both $xy$ and $yx$ start to increase again when $K > 10{,}000$, at which point we start having significant replication (observe the average $x$- and $y$-extent statistics in Table 4.1). Figure 4.2 breaks down the total cost to partitioning and joining for the $xy$ case. The joining cost includes the cost of sorting the partitions. As expected, the cost of partitioning increases with $K$ and the joining cost drops. After $K = 10{,}000$ partitioning becomes very expensive without offering improvement in the join. The lowest runtime is achieved when the $x$-extent of the partitions (i.e., the narrow side of the stripes) is about 10 times larger than the average $x$-extent of the rectangles. In this case, the chances that a rectangle is replicated to neighboring stripes are small and at the same time, the stripes are narrow enough for plane sweep to be effective (i.e., the chance that a candidate pair that $x$-intersects also $y$-intersects is not low). For the rest of our analysis we use $xy$ as the default setup for 1D partitioning.

**Tuning 2D Partitioning.** We now repeat the same set of experiments from the previous section, but this time using a 2D partitioning. We vary the granularity $K \times K$ of the grid and measure for each value of $K$ the runtime cost of the algorithm, when the sweeping axis is always set to $x$, always set to $y$, or when our adaptive model is used to select the sweeping axis at each tile (which could be different at different tiles). Figure 4.3 depicts the performance of the three join variants. Similarly to 1D partitioning, when the number of partitions is small $K \leq 20$, the choice of the sweeping axis makes a difference and choosing $x$ is better. In these configurations, our model can be even better than always choosing $x$. The three options converge at about $K = 500$. Figure 4.4 shows the cost breakdown for the partitioning and joining phases of the 2D spatial join, when our model is used for picking the sweeping axis $x$. The observations are similar the corresponding ones for 1D partitioning. We observe that the cost of partitioning increases with $K$ and becomes too high when the tiles become too many and very small (i.e., when $K > 2{,}000$). On the other hand, the




(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 4.3: Tuning 2D partitioning: total execution time




(a) $T2 \bowtie T5$ (b) $O6 \bowtie O9$

Figure 4.4: Tuning 2D partitioning: time breakdown

Table 4.3: 1D vs. 2D partitioning: speedup

| query | 1D | | 2D | |
|---|---|---|---|---|
| | $K$ | speedup | $K \times K$ | speedup |
| $T2 \bowtie T5$ | 3000 | 9.6x | $1000 \times 1000$ | 8.16x |
| $T2 \bowtie T8$ | 7000 | 10.67x | $2000 \times 2000$ | 8.98x |
| $O5 \bowtie O6$ | 3000 | 8.62x | $1000 \times 1000$ | 6.82x |
| $O6 \bowtie O9$ | 7000 | 16.56x | $2000 \times 2000$ | 12.58x |

join cost drops, but stabilizes after $K > 2{,}000$. After this point, the number $K \times K$ of tiles (that have to be managed) becomes significantly high and replication becomes excessive. The joining phase does not benefit; due to replication, the join inputs at each tile do not reduce in size and the same join results are computed in neighboring tiles. In addition, our tests show that the 2D partitioning version of the algorithm should always use our adaptive model to select the sweeping axis.

**1D vs. 2D Partitioning.** There are two main findings from the PBSM tuning experiments. First, the rule of the thumb is to select $K$ (in both 1D and 2D partitioning) such that the extents of the resulting partitions are about one order of magnitude larger than the extents of the rectangles (in one or both dimensions, respectively). Second, 1D partitioning achieves better performance compared to 2D partitioning, due to less replication and the fact that all tiles in a row or a column can be swept by a single line (along the row or column) with the same effect as processing all tiles independently with sweeping along the same direction. Table 4.3 summarizes, for the four join queries, the best speedups achieved by 1D and 2D partitioning, compared to the best corresponding performance of the plane sweep algorithm without partitioning. 1D partitioning is up to 32% faster compared to 2D partitioning.

#### 4.1.3.4 Parallel Evaluation

Last, we test the parallel version of the algorithm using 1D partitioning. Table 4.4 summarizes, for the four join queries, the runtime and the speedup achieved by our parallel evaluation of the spatial join. The performance scales gracefully with the number of threads, until it stabilizes over 20 threads, which equals the number of physical cores in our machine. As a general conclusion, our parallel design takes full advantage of the system resources to greatly reduce the join cost.

Table 4.4: Parallel evaluation: runtime (1D partitioning)

| # threads | queries | | | |
|---|---|---|---|---|
| | $O5 \bowtie O6$ | $O6 \bowtie O9$ | $T4 \bowtie T8$ | $O9 \bowtie O3$ |
| 1 | 2.98s | 14.4s | 20.1s | 43.0s |
| 5 | 0.75s | 3.32s | 4.34s | 10.6s |
| 10 | 0.46s | 1.91s | 2.47s | 6.11s |
| 15 | 0.38s | 1.45s | 1.85s | 4.54s |
| 20 | 0.32s | 1.21s | 1.64s | 3.54s |
| 25 | 0.29s | 1.07s | 1.42s | 3.09s |
| 30 | 0.28s | 0.99s | 1.36s | 2.89s |
| 35 | 0.27s | 0.96s | 1.27s | 2.72s |
| 40 | 0.27s | 0.91s | 1.21s | 2.72s |

## 4.2 A Two-layer Partitioning for Non-point Spatial Data

### 4.2.1 Two-layer Spatial Partitioning

In this section, we present our secondary partitioning approach for (in-memory) SOP spatial indices. We consider non-point spatial objects, indexed using their MBR approximations. Even though our approach can be used in any SOP index, we will present it in the context of a grid index, Consider a regular $N \times M$ regular grid, which primarily divides the space into $N \cdot M$ disjoint spatial partitions, called *tiles*. An object $o$ is assigned to a tile $T$ iff $MBR(o)$ and $T$ intersect (i.e., they have at least one common point); in this case, $o$ is assigned to tile $T$. Since $MBR(o)$ can intersect with multiple tiles, $o$ can be assigned to more than one tiles. For example, Figure 2.1 shows a grid and a spatial object $o_1$, (colored darkgrey), whose MBR intersects tiles $T_a$ and $T_b$; $o_1$ is assigned to both tiles. For each tile $T$, we keep a list of (MBR, object-id) pairs that are assigned to $T$. This means that while the MBRs and ids of the objects can be replicated to multiple tiles, the actual geometry of an object is stored only once in a separate data structure (e.g., an array or a hash-map) in order to be retrieved fast, given the object's id.

Since the spatial distribution of objects may not be uniform, there could be empty tiles. If the percentage of empty tiles is very large, to save memory, we can use a hash-table to map each non-empty tile to the set of rectangles assigned to it. The above storage scheme is quite effective for main-memory data because it supports queries

and updates quite fast, while it is straightforward to parallelize popular spatial queries and operations.

**Secondary Partitioning.** We propose that the set of MBRs at each tile is further divided into four classes $A$, $B$, $C$, and $D$ (which are physically stored separately in memory). Recall that each MBR $r$ can be represented by an interval of values at each dimension. Let $r.x = [r.x_l, r.x_u]$ be the projection of rectangle $r$ on the $x$ axis and $r.y = [r.y_l, r.y_u]$ $r$'s $y$-projection. Now, consider a rectangle $r$ which is assigned to tile $T$.

- $r$ belongs to class $A$, if for every dimension $d \in \{x, y\}$, the begin value $r.d_l$ of $r$ falls into projection $T.d$, i.e., if $T.d_l \leq r.d_l$.
- $r$ belongs to class $B$ if $r.x$ begins inside $T.x$ and $r.y$ begins before $T.y$, i.e., if $T.x_l \leq r.x_u$ and $T.y_l > r.y_l$.
- $r$ belongs to class $C$ if $r.x$ begins before $T.x$ and $r.y$ begins inside $T.y$, i.e., if $T.x_l > r.x_l$ and $T.y_l \leq r.y_l$.
- $r$ belongs to class $D$ if both its $x$- and $y$-projections begin before $T$, i.e., if $T.x_l > r.x_l$ and $T.y_l > r.y_l$.

Figure 4.5 illustrates examples of rectangles in a tile $T$ that belong to the four different classes.[2] During data partitioning, for each tile $T$ a rectangle $r$ is assigned to, we identify its class and place it to the corresponding division. Note that a rectangle can belong to class $A$ of just one tile, while it can belong to other classes (in other tiles) an arbitrary number of times. We denote the secondary partitions of tile $T$ which store the MBRs of classes $A$, $B$, $C$, and $D$, by $T^A$, $T^B$, $T^C$, and $T^D$, respectively.

### 4.2.2 Range Query Evaluation

In this section, we show how the secondary partitions at each tile $T$ can be used to evaluate spatial range queries efficiently and at the same avoid generating and identifying duplicate query results. We first consider rectangular range queries $W$ (window queries). The cases of other query shapes will be discussed later on. For now, we focus on the *filtering step* of the query, i.e., the objective is to just find the

[2] We conventionally assume that the $x$ dimension is from left to right and the $y$ dimension is from top to bottom.

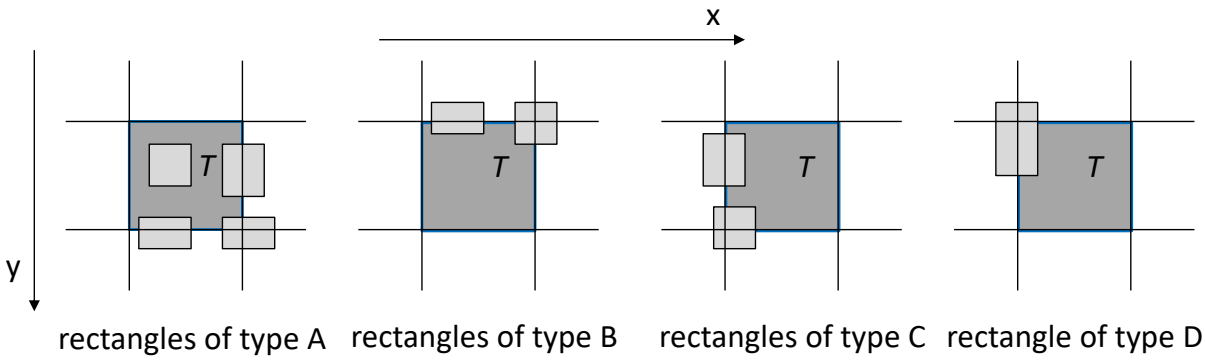



Figure 4.5: The four classes of rectangles assigned to a tile $T$.

object MBRs which intersect $W$. The refinement step will be discussed in Section 4.2.3.

First, the tiles in a $N \times M$ grid, which intersect $W$ can be easily found by algebraic operations. Specifically, assuming that tile $T_{i,j}$ is at the $i$-th row and at the $j$-th column of the grid, the tiles which intersect $W$ are all tiles $T_{i,j}$, for which $\lfloor W.x_l/N \rfloor \leq i \leq \lfloor W.x_u/N \rfloor$ and $\lfloor W.y_l/M \rfloor \leq j \leq \lfloor W.y_u/M \rfloor$. We now explain in detail, for each tile $T$ that intersects $W$, which classes of rectangles should be accessed and which computations are necessary for determining whether each rectangle $r$ intersects $W$. Our goal is not only to avoid accessing irrelevant secondary partitions, but also to minimize the computational cost for finding the query results in the relevant partitions of $T$.

#### 4.2.2.1 Selecting relevant classes

For a tile $T$, let $prev(T, d)$ denote the tile which is right before $T$ in dimension $d$ and has exactly the same projection as $T$ in the other dimension. For example, in Figure 4.6, $prev(T, x)$ (resp. $prev(T, y)$) is the tile right before $T$ in dimension $x$ (resp. $y$). Given a window query $W$, the following lemmas determine the classes of rectangles in $T$ which should be disregarded, because they can only produce duplicate results.

**Lemma 4.1.** *If the query range $W$ intersects tile $T$ and $W$ starts before $T$ in dimension $x$, then secondary partitions $T^C$ and $T^D$ should be disregarded.*

*Proof.* Consider a rectangle $r$ in class $C$ or class $D$ of tile $T$, i.e., $r \in T^C$ or $r \in T^D$. Rectangle $r$ should also be assigned to the previous tile $prev(T, x)$ to $T$ in dimension $x$, because it belongs to class $C$ or $D$ of $T$. If $r$ intersects $W$ in $T$, then $r$ should also intersect $W$ in $prev(T, x)$, because $W$ also starts before $T$ in dimension $x$. Hence,

examining and reporting $r$ in tile $T$ would produce a duplicate, since the same result can also be identified in tile $prev(T, x)$. □

**Lemma 4.2.** *If $W$ intersects tile $T$ and $W$ starts before $T$ in dimension $y$, then secondary partitions $T^B$ and $T^D$ should be disregarded.*

Lemma 4.2 can be proved by replacing $x$ by $y$ and $C$ by $B$ in the proof of Lemma 4.1. The two lemmas are combined to exclude all classes $B$, $C$, and $D$ if $W$ starts before $T$ in both dimensions. To illustrate the lemmas, consider tile $T$ in Figure 4.6.

Consider the MBRs of objects $o_1$ and $o_2$, which belong to secondary partitions $T^B$ and $T^C$, respectively. $MBR(o_1)$ should be ignored when processing $T$ because it belongs to class $B$ and $W$ starts before $T$ in dimension $y$ (Lemma 4.2). Indeed, $MBR(o_1)$ also intersects $W$ also in tile $prev(T, y)$ which is right above $W$. On the other hand, $W$ does not start before $T$ in dimension $x$, i.e., Lemma 4.1 does not apply for tile $T$. This means that $MBR(o_2) \in T^C$ will be found to intersect $W$. Figure 4.6 shows, in the top-left corner of each tile $T$ intersected by $W$, the object classes in $T$ that should be examined (the remaining classes can be disregarded). Observe that we have to consider all objects in just one tile (the one containing point $(W.x_l, W.y_l)$). For the majority of tiles, we only have to examine secondary partition $T^A$.

#### 4.2.2.2 Minimizing the comparisons

We now turn our attention to *minimizing the comparisons* needed for each secondary partition that *has to* be checked (i.e., those not eliminated by Lemmas 4.1 and 4.2). Recall that for a rectangle $r$ in a tile $T$ to intersect the query window $W$, $r$ should intersect $W$ in all dimensions. To verify this, we need at most four comparisons (i.e., if $r.x_u < W.x_l$ or $r.x_l > W.x_u$ or $r.y_u < W.y_l$ or $r.y_l > W.y_u$, then $r$ and $W$ do not intersect).

A direct observation that saves comparisons is that, if a tile $T$ is covered by the window $W$ in a dimension $d$, then we do not have to perform intersection tests in dimension $d$ for all rectangles in the relevant secondary partitions in $T$. In the example of Figure 4.6, we need to examine partitions $T^A$ and $T^C$ of tile $T$ (Lemma 4.2). For each rectangle $r$ in these partitions, we only have to verify if projections $r.x$ and $W.x$ intersect, because $r.y$ and $W.y$ definitely intersect (since $T.y$ is covered by $W.y$).

For the dimension(s) where $T$ is not covered by $W$, the following lemmas can be used to further reduce the necessary comparisons.

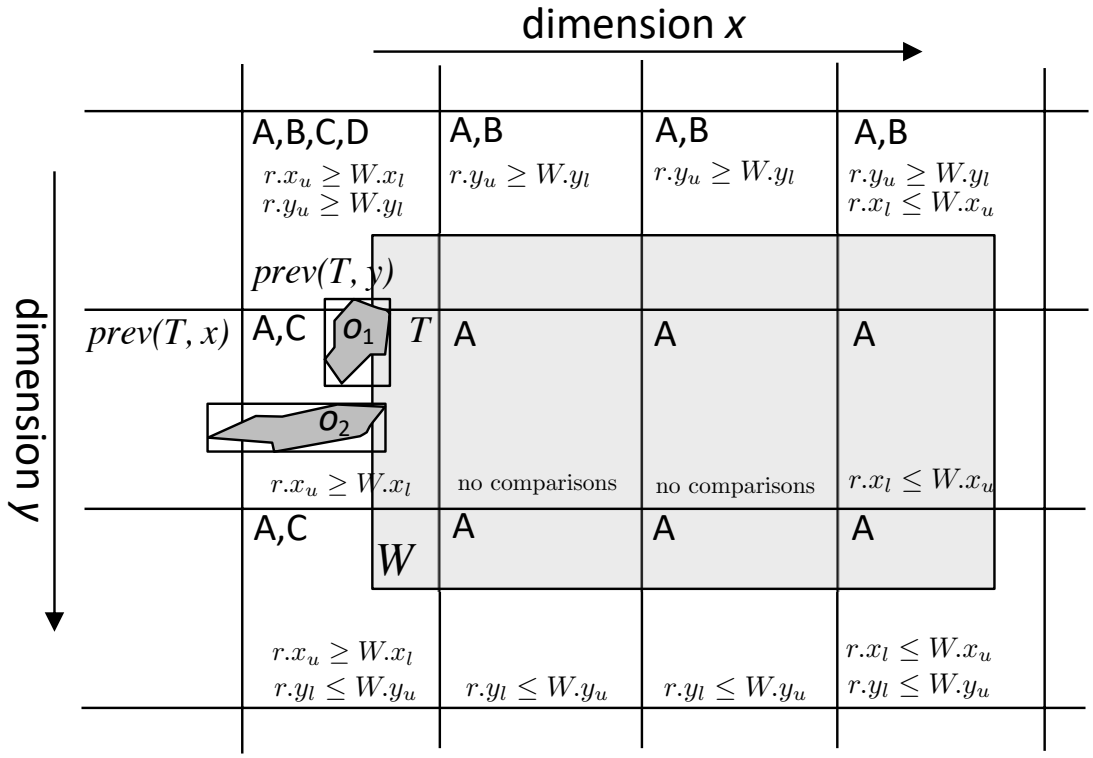



Figure 4.6: Examples of object classes and comparisons

**Lemma 4.3.** *If $W$ ends in tile $T$ and starts before $T$ in dimension $d$, then for a rectangle $r \in T$, $r$ intersects $W$ in dimension $d$ iff $r.d_l \leq W.d_u$.*

Symmetrically, we can show:

**Lemma 4.4.** *If $W$ starts in tile $T$ and ends after $T$ in dimension $d$, then for a rectangle $r \in T$, $r$ intersects $W$ in dimension $d$ iff $r.d_u \geq W.d_l$.*

For example, in tile $T$ of Figure 4.6, we only have to test intersection in dimension $x$, as already explained. The intersection test can be reduced to a simple comparison, i.e., $r$ intersects $W$ iff $r.x_u \geq W.x_l$, due to Lemma 4.4.

To demonstrate the impact of Lemmas 4.3 and 4.4, in each tile of the figure, we show the necessary comparisons. For the two tiles in the center, no comparisons are required because all MBRs (in class A) are guaranteed to intersect $W$. For the remaining two tiles, which intersect the border of $W$, we only have to perform at most one comparison per dimension, because $W$ either starts or ends at these tiles (and some of these tiles are totally covered by $W$ in one dimension). Contrast this to the four comparisons required in the general case for testing whether two rectangles (e.g., $r$ and $W$) intersect. Therefore, for range queries that cover multiple tiles, we have:

**Corollary 4.1.** *For a window query $W$ that intersects more than one tile per dimension, the number of required comparisons per rectangle in each relevant tile is at most two.*

#### 4.2.2.3 Storage decomposition

In order to further reduce the number of comparisons and improve the data access locality, we suggest to store the MBRs of each one of the secondary partitions $\{T^A, T^B, T^C, T^D\}$ in *decomposed tables*, following the Decomposition Storage Model (DSM) [162], adopted by column-oriented database systems (e.g., [163]). Specifically, each rectangle $r = \langle id, r.x_l, r.x_u, r.y_l, r.y_u \rangle$ is decomposed to four tuples, i.e., $\langle r.x_l, id \rangle$, $\langle r.x_u, id \rangle$, $\langle r.y_l, id \rangle$, $\langle r.y_u, id \rangle$, and each tuple is stored in a dedicated table, i.e., $L_{x_l}, L_{x_u}, L_{y_l}, L_{y_u}$. The tables are sorted by their first column and used to evaluate fast queries on tiles, where just one endpoint of each MBR needs to be compared (according to Lemmas 4.3 and 4.4). We can take advantage of the sorted decomposed tables to reduce the information that has to be accessed and the number of comparisons.

In particular, for each tile $T$ satisfying Lemma 4.3 in dimension $d$, we can perform *binary search* on the table $L_{d_l}$ which stores the $\langle r.d_l, i \rangle$ tuples to find the largest $r.d_l$, which satisfies $r.d_l \leq W.d_u$. All rectangles in the table up to this value are guaranteed to satisfy the condition and can be reported without any comparison.[3] Symmetrically, we can reduce the comparisons for rectangles in a tile $T$, which satisfies Lemma 4.4, by taking advantage of the sorted table $L_{d_u}$. For example, for the tile $T$ in Figure 4.6, we only have to access and perform binary search to tables $L^A_{x_u}$ and $L^C_{x_u}$, which store the $\langle r.d_l, i \rangle$ decompositions of the rectangles in secondary partitions $T^A$ and $T^C$, respectively. If we have to perform two comparisons in a tile (e.g., $r.x_u \geq W.x_l$ and $r.y_u \geq W.y_l$), we choose one of the two relevant decomposed tables (e.g., $L_{x_u}$ or $L_{y_u}$) to perform the search; then, for each qualifying rectangle according to the selected comparison, we verify the other comparison by accessing the entire MBR. We select the table in the dimension which is covered the least by $W$, in order to minimize the necessary verifications.

Finally, we observe that, for some object classes, it is not necessary to store all decompositions. For example, the only possible comparisons that can be applied to rectangles of class $D$ are $r.x_u \geq W.x_l$ and $r.y_u \geq W.y_l$, because all MBRs of class $D$ start before the tile in both dimensions and they are only compared with $W$ in the

[3] Alternatively, we can scan from the beginning of the table until the condition is violated.

tile that includes the start point of $W$ in both dimensions (Lemma 4.4). Hence, we only need to keep tables $L^D_{x_u}$ and $L^D_{y_u}$ for each secondary partition $T^D$. Overall, we can reduce the storage requirements for the decomposed tables as shown in Table 4.5.

Table 4.5: Required decomposed tables for each secondary partition

| **partition** | **required tables** |
|---|---|
| $T^A$ | $L^A_{x_l}$, $L^A_{x_u}$, $L^A_{y_l}$, $L^A_{y_u}$ |
| $T^B$ | $L^B_{x_l}$, $L^B_{x_u}$, $L^B_{y_u}$ |
| $T^C$ | $L^C_{x_u}$, $L^C_{y_l}$, $L^C_{y_u}$ |
| $T^D$ | $L^D_{x_u}$, $L^D_{y_u}$ |

The decomposed data representation not only reduces the number of comparisons but also accesses only the necessary data for each verified comparison. In particular, rectangle coordinates which are not relevant to the required verification are not accessed at all, while in a record-based representation irrelevant data are fethed to the memory cache. On the other hand, the decomposed representation requires additional storage and is more expensive to update (unless a batch update strategy is employed); hence, it is mostly appropriate for indexing static spatial object collections.

#### 4.2.2.4 Overall approach

Algorithm 4.1 describes the steps of window query evaluation. Given a window $W$, we first identify the range of tiles that intersect $W$ by simple algebraic operations, as discussed in the beginning of this section. We then pass the control to each relevant tile $T$, which accesses the relevant secondary partitions and performs the necessary computations for the rectangles in them, potentially using the decomposed tables presented in Section 4.2.2.3. Note that the operations at each tile $T$ (and each secondary partition in $T$) are *totally independent* to each other and they can be parallelized without the need of any synchronization.

#### 4.2.2.5 Non-rectangular ranges

Window queries are the most popular range queries. Still, not all query ranges are rectangular. We now discuss the evaluation of non-rectangular range queries. A char-

**Algorithm 4.1** Window query evaluation (filtering step)

**Require**: grid $\mathcal{G}$, query window $W$

1: $\mathcal{T}$ = tiles in $\mathcal{G}$ that intersect $W$
2: **for** each tile $T \in \mathcal{T}$ **do**
3: $\quad \mathcal{P}_T$ = secondary partitions of $T$ relevant to $W$
4: $\quad$ **for** each partition $T^X \in \mathcal{P}_T$ **do**
5: $\quad\quad$ find all $r \in T^X$ that intersect $W$
6: $\quad$ **end for**
7: **end for**

acteristic non-rectangular range query is the *disk* (or *distance*) range query, where the objective is to find all objects with (minimum) distance to a given query point $q$ at most $\epsilon$. To evaluate a disk query on our two-layer partitioned dataset, we apply a similar method to Algorithm 4.1; we first find the set of tiles $\mathcal{T}$ that intersect with the disk (using algebraic/trigonometric operations) and then find the objects in them that satisfy the query predicate. We now discuss how our secondary partitioning can be used to reduce the number of considered rectangles by tile and avoid duplicates at the same time. As in window queries, for each tile $T \in \mathcal{T}$, we check whether $prev(T, d)$ in each dimension $d$ is also in $\mathcal{T}$. If yes, then we disregard the corresponding class of rectangles in $T$. Hence, if $prev(T, x) \in \mathcal{T}$, then classes $B$ and $D$ are disregarded, whereas if $prev(T, y) \in S$, then classes $C$ and $D$ are disregarded. Figure 4.7 shows an example of a disk query centered at q. The tiles which intersect the disk are shown by different patterns depending on the classes of rectangles in them that have to be checked. For example, in tile $T_5$ all four classes will be examined (we call $T_5$ an $ABCD$ tile, in the context of the disk query). Note that for the majority of tiles which intersect the disk range, we only have to examine rectangles in class $A$.

A subtle point here is that if we simply examine all rectangles in the classes that correspond to each tile, we may end up examining duplicates. For example, consider rectangle $r_1$, which will be examined in both tiles $T_1$ (in class $B$) and $T_2$ (in class $C$). To avoid such duplicates, for each rectangle in an $ABCD$ tile $T$, if the tile is closer to $q$ in the $y$-dimension compared to the $x$-dimension, we ignore rectangles $r$ in classes $C$ and $D$, for which $r.y_u > T.y_u$ (these will be handled in another tile). The case where $T$ is closer to $q$ in the $x$-dimension is handled symmetrically.

For tiles which are totally covered by the disk range, we do not verify any dis-

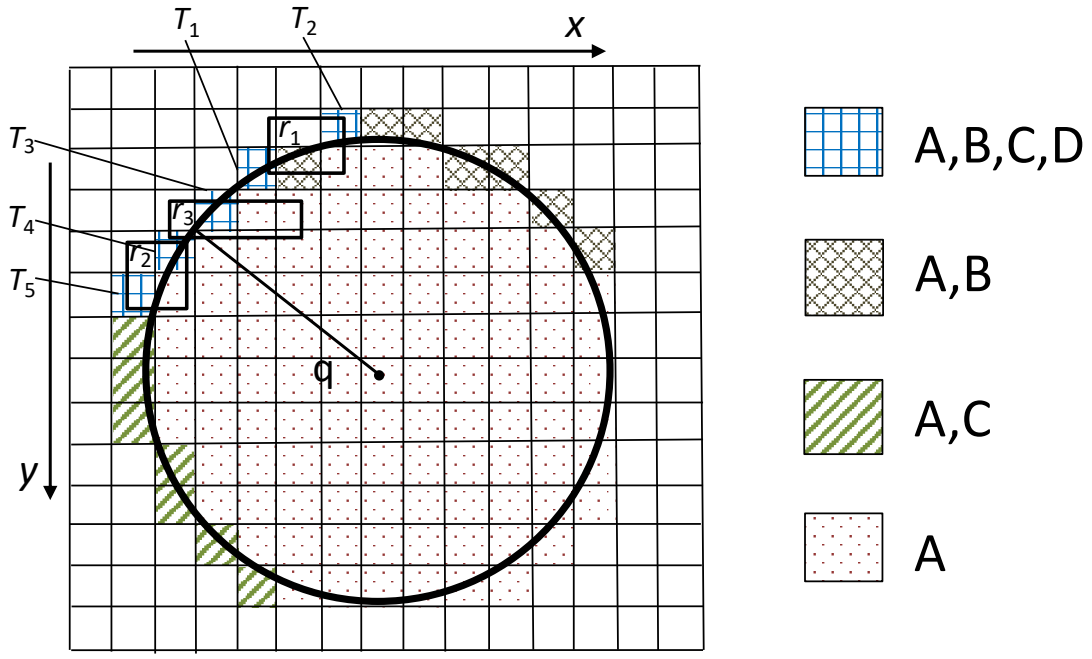



Figure 4.7: Example of disk query evaluation

tances between the objects assigned to them and $q$, as these are guaranteed to be query results. Distance verification only has to be performed for objects in tiles which partially intersect the disk.

The method described above for disk queries can be generalized for any non-rectangular query. We first find the set of tiles $\mathcal{T}$ which intersect the query range. Then, for each tile $T \in S$, we determine which classes of objects need to be examined (i.e., exclude classes that would produce duplicates). For each tile which is totally covered by the query region, we just report its contents in the relevant classes as results and for the remaining tiles we conduct an intersection test for each rectangle before determining whether it is a result.

### 4.2.3 Refinement Step

We now discuss the evaluation of the refinement step of range queries using our secondary partitioning scheme. We show how we can drastically reduce the number of objects for which a refinement step is necessary and at the same time unveil that the refinement step is not the bottleneck in range query evaluation. This justifies why our focus is on the filter step. We begin by a general and important lemma, which is independent to our approach and then show how our secondary partitioning can further accelerate the identification of objects for which a refinement step is not

necessary.

**Lemma 4.5** (Secondary filtering). *Given a candidate object whose MBR $r$ intersects the query range, if at least one side of $r$ is inside the query range, then the object is guaranteed to intersect the range and no refinement step for the object is necessary.*

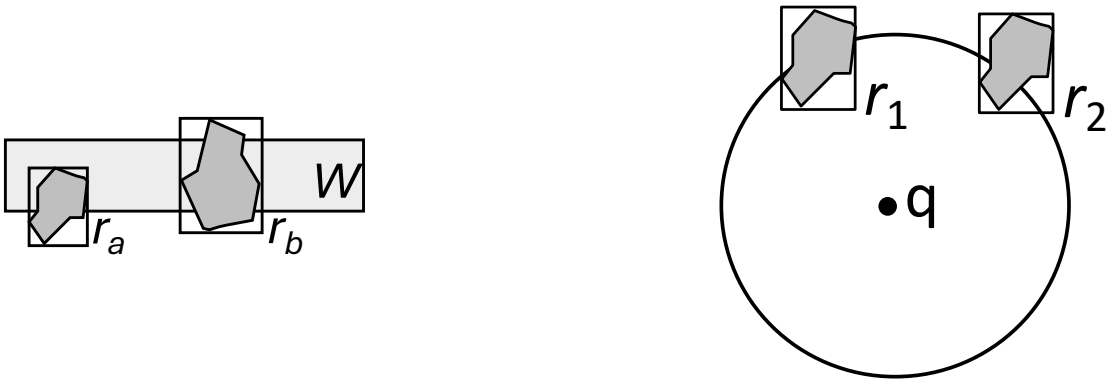
The lemma is trivial to prove, based on the definition of MBR. Recall that the MBR of an object is defined by the minimum and maximum values of the object in every dimension. Hence, at each side of the MBR, there is at least one point which is part of the object's geometry. If one side of the MBR is inside the query range, then there should be at least one point of the object inside the query range, i.e., the object and the range intersect. The lemma generalizes to more than two dimensions. In a $d$-dimensional space, we test if at least one of the $(d-1)$-dimensional faces of the minimum hypercube that bounds the object is inside the query range. For rectangular query ranges $W$, we can simplify the test by checking whether at least one of the projections $r.x$ or $r.y$ of $r$ is covered by the corresponding projection $W.x$ or $W.y$ of $W$. If this is true, given that $r$ intersects $W$, at least one point of the object corresponding to $r$ should be inside $W$. This test costs at most four comparisons.

For example, in Figure 4.8a, either one side of $r$ is inside the window $W$ and Lemma 4.5 applies (see $r_a$ in Figure 4.8a) or $r$ splits $W$ along th e coverage axis (see $r_b$ in Figure 4.8a). In this case an edge of the rectangle is not contained in $W$, but $W$ splits the MBR along the coverage axis; it is not possible that the object in the MBR does not intersect $W$, unless the area that forms the object is not connected. In both cases, whatever the geometry of the object is, the object definitely intersects $W$.

For a disk query range, we can check whether there are at least two corners of $r$ whose distances to the disk center are smaller than or equal to the disk radius (in this case at least one side of $r$ should be inside the disk). This test costs at most four distance computations.

For example, in Figure 4.8b, rectangle $r_1$ has at least two corners in the disk range, which means that at least one side of the rectangle is in the range and Lemma 4.5 applies. On the other hand, only one corner of $r_2$ is inside the disk, hence the refinement step for the corresponding object cannot be avoided.

**Efficient secondary filtering.** We now show how we can use our two-layer partitioning approach to reduce the cost of applying the post-filtering tests (Lemma 4.5). The main idea is to study the refinement avoidance test at the *tile level*, in order to




(a) window query (b) disk query

Figure 4.8: Secondary filtering for range queries

limit the comparisons required for each class of objects in the tiles. Specifically, for each $T$ that intersects a query range $W$ and for each dimension $d$, we consider two cases: (i) $W$ starts before $T$ in dimension $d$, i.e., $W.d_l < T.d_l$ and (ii) $W.d_l \geq T.d_l$. In the first case, due to Lemmas 4.1 and 4.2, only classes of rectangles that start inside $T$ in dimension $d$ are considered, which means for each rectangle $r \in T$ which is found to intersect $W$, we already know that $W.d_l < r.d_l$. Hence, we only have to test if $r.d_u \leq W.d_u$ to confirm whether $r$ is covered by $W$ in dimension $d$. On the contrary, for the case where $W.d_l \geq T.d_l$, we should apply the complete coverage test (i.e., $W.d_l \leq r.d_l \wedge r.d_u \leq W.d_u$) in dimension $d$.

For example, in Figure 4.6, we only have to perform the complete coverage test for all rectangles that intersect $W$ in the top-left tile ($prev(T, y)$). In the tiles, where we access classes $A$ and $B$, we save one comparison in the $x$ dimension, in the tiles, where we access classes $A$ and $C$, we save one comparison in the $y$ dimension, and in all other tiles (where we access only class $A$), we save one comparison per dimension.

### 4.2.4 Batch Query Processing

In the previous sections, we presented how our two-layer index handles single query requests. Real systems however receive and need to evaluate a large number of concurrent queries. Under this, we next discuss how to efficiently process batches of spatial range queries. Although our focus is primarily in a single-threaded processing environment, parallel query processing in modern multi-core hardware can also benefit from the ideas discussed in this section. To this end, our experimental analysis

includes both single-threaded and multi-threaded experiments.

**Queries-based approach.** A straightforward approach for processing a workload of concurrent spatial range queries is to directly evaluate every query independently.

In a parallel processing environment, we can easily adopt this approach by assigning the queries to the available threads in a round robin fashion. We call this simple approach queries-based. Its main shortcoming is that it is cache agnostic; as every issued query $q$ typically overlaps multiple tiles of the grid, the computation of $q$ requires accessing data structures in different parts of the main memory, i.e., the memory access pattern is prone to cache misses. The problem is present also in parallel query processing, as every thread goes through multiple rounds of *content switching*.

**Tiles-based approach.** To address this shortcoming of queries-based, we design a cache-conscious two-step approach. Given a large batch of queries $Q$, for each tile, accumulate the subtasks of all queries in $Q$ that intersect the tile. Each subtask corresponds to accessing and processing (the relevant to the query) secondary partitions in the tile. Then, in a second step, we initiate one process at each tile, which evaluates the corresponding subtasks. Essentially, query processing is no longer driven by the queries, but from the grid tiles and therefore, we call this approach tiles-based. This method is favored by parallel processing, since each thread (corresponding to a tile) can benefit from the processor's cache while processing the subtasks assigned to it. As we demonstrate in Section 4.2.6 the tiles-based approach scales better with the number of parallel threads compared to queries-based.

### 4.2.5 Spatial Join Evaluation

We now turn our focus to the evaluation of spatial intersection joins. First, we discuss how our two-layer partitioning can be adopted to natively compute a spatial join, while avoiding the generation and elimination of duplicate results. To this end, we consider the basic variant of our partitioning, described in Section 4.2.1 without the storage decomposition in Section 4.2.2.3. Then, we elaborate on possible join strategies for a query optimizer perspective, which use two-layer partitioning in different fashions. For illustration purposes, we discuss the above for regular grids; in Section 4.2.5.3, we consider other SOPs, e.g., the quad-tree.

#### 4.2.5.1 Two-layer Partitioning Join

Assume that both join input datasets $R$, $S$ are indexed by our two-layer partitioning scheme, specifically 2-layer and that the two grids used for the partitioning are identical. In the next subsection, we discuss the origins and details of such a setting, i.e., whether both or one of $R$, $S$ are (re)-indexed online. Under this setting, we can build upon the join phase of the PBSM algorithm [14] and apply a partition-to-partition join for each pair of partitions from $R$, $S$ from the same tile. Assuming that two partitions of size $n$, $m$ are joined, the join cost using plane-sweep is $O((n+m)\log(n+m))$, based on [18, 81].

##### 4.2.5.1.1 The Mini-joins Breakdown

Given a tile $T$, let $R_T$ and $S_T$ be the partitions containing the object rectangles from datasets $R$ and $S$, respectively, that are assigned to $T$. $R_T$ is divided into rectangle classes $R_T^A$, $R_T^B$, $R_T^C$, and $R_T^D$, according to our two-layer partitioning scheme discussed in Section 4.2.1. Similarly, $S_T$ is divided into $S_T^A$, $S_T^B$, $S_T^C$, and $S_T^D$. Hence, the spatial join $R_T \bowtie S_T$ can now be decomposed into $4{\cdot}4 = 16$ joins between classes of rectangles, i.e., $R_T^A \bowtie S_T^A, R_T^B \bowtie S_T^B, R_T^B \bowtie S_T^C, \ldots, R_T^B \bowtie S_T^A, \ldots$. We call these class-to-class joins, *mini-joins*. Figure 4.9 exemplifies the decomposition of the partition-to-partition join inside a tile $T$ into the 16 mini-joins.

Looking deeper into these 16 cases, we can easily show that 7 out of these mini-joins (the shaded cases in the figure) produce only duplicate results, i.e., join results that will also be reported in another tile. For example, if two rectangles of class $B$ in tile $T$ intersect (i.e., the pair is contained in the result of $R_T^B \bowtie S_T^B$), then they will definitely intersect also in another tile above $T$. We can also show that the remaining 9 mini-joins produce only results that cannot be reported in any previous tile (in the $x$ or $y$ dimension or in both), but could be produced as duplicates in some of the 7 shaded joins in a tile *after* $T$ in one or both dimensions. Hence, we *never* evaluate the 7 shaded mini-joins and only evaluate the 9 remaining without the need of duplicate elimination.

##### 4.2.5.1.2 Optimizations

Any spatial join algorithm can be utilized to evaluate the set of 9 mini-joins on each tile $T$; even a nested-loops approach. Following previous work on spatial joins

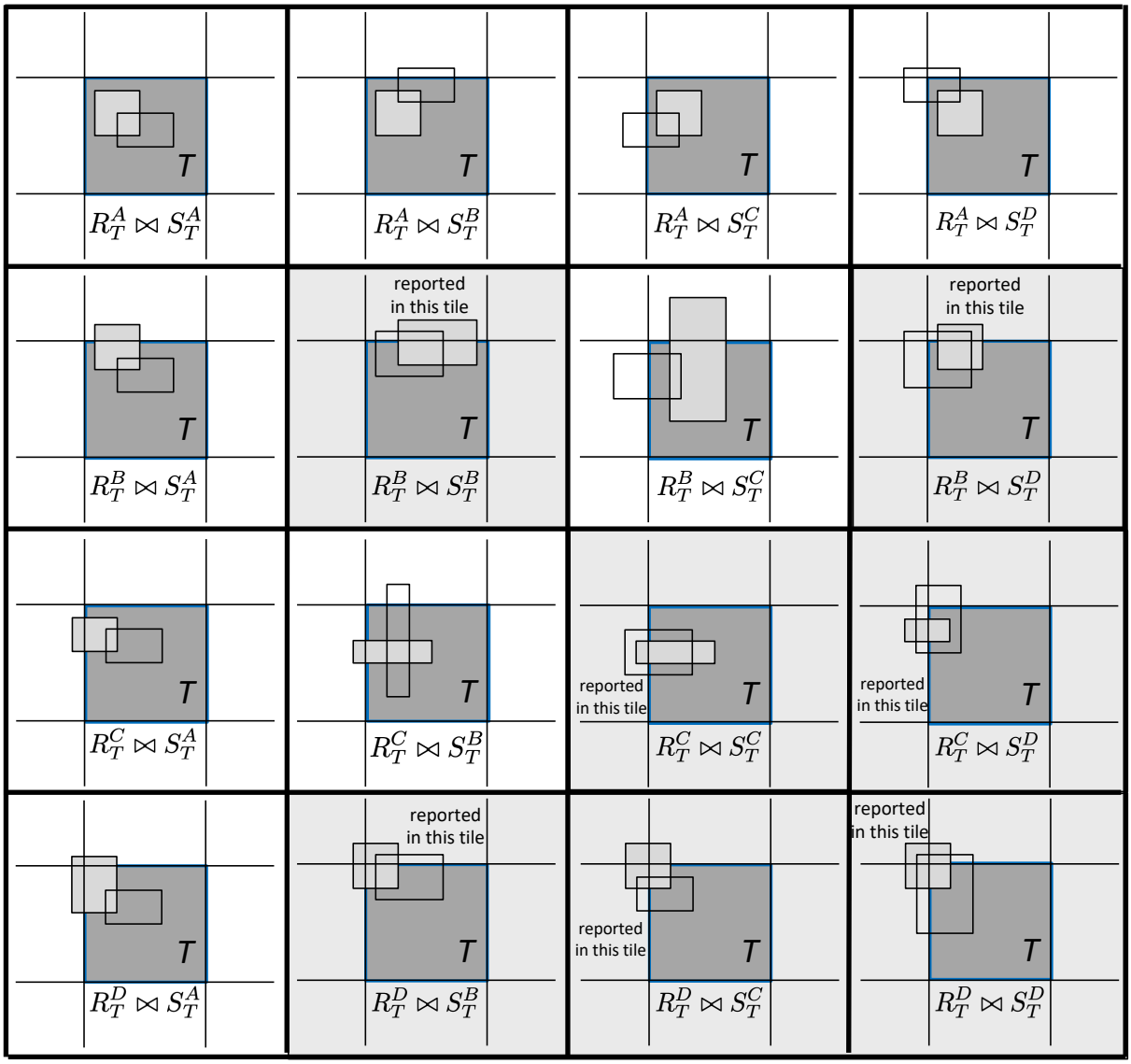



Figure 4.9: Decomposition of tile $T$'s join into 16 mini-joins; $R$ rectangles filled in light gray color.

[164, 14, 81], we adopt a plane-sweep join approach which is shown to perform significantly faster than nested-loops; specifically we adopt the plane-sweep algorithm in [18]. Algorithm 4.2 illustrates our plane-sweep mini-join. To apply the plane-sweep approach, the contents of each class are sorted on the sweeping dimension (Line 1). Without loss of generality, we consider $x$ as the sweeping dimension for the rest of this section, and so rectangles are sorted by $r.x_l$ and $s.x_l$. Note that this sorting can take place also during the construction of the two-layer scheme, if an input dataset is partitioned online. For every pair $(r, s)$ determined by the sweeping process (i.e., rectangles whose projections on $x$ intersect), we test in Lines 6 and 14,

**Algorithm 4.2** Plane-sweep mini-join

**Require**: classes of rectangles $R_T$ and $S_T$

1: **sort** $R_T$ and $S_T$ by $r.x_l$ ▷ `if not already sorted`
2: **while** $R_T$ and $S_T$ not depleted **do**
3: **if** $r.x_l < s.x_l$ **then**
4: $s' \leftarrow s$
5: **while** $s' \neq$ null **and** $r.x_u \geq s'.x_l$ **do**
6: **if** $r.y_l \leq s'.y_l \leq r.y_u$ **or** $s'.y_l \leq r.y_l \leq s'.y_u$ **then**
7: **output** $(r, s')$ ▷ `update result`
8: **end if**
9: $s' \leftarrow$ next rectangle in $S_T$ ▷ `scan forward`
10: **end while**
11: **else**
12: $r' \leftarrow r$
13: **while** $r' \neq$ null **and** $s.x_u \geq r'.x_l$ **do**
14: **if** $r'.y_l \leq s.y_l \leq r'.y_u$ **or** $s.y_l \leq r'.y_l \leq s.y_u$ **then**
15: **output** $(r', s)$ ▷ `update result`
16: **end if**
17: $r' \leftarrow$ next rectangle in $R_T$ ▷ `scan forward`
18: **end while**
19: **end if**
20: **end while**

if the rectangles also intersect in the second dimension $y$; i.e., if $r.y_l \leq s.y_l \leq r.y_u$ or $s.y_l \leq r.y_l \leq s.y_u$. To boost the computation of mini-joins, we next discuss how we can save on the comparisons performed for $(r, s)$ pairs, capitalizing on our second layer of partitioning.

**Avoid unnecessary comparisons.**

There exist two ways to utilize the $A$, $B$, $C$, $D$ classes in each tile $T$ for avoiding unnecessary rectangle comparisons. To understand the first, consider the $R_T^A \bowtie S_T^C$ mini-join. By definition, all objects rectangles in $S_T^C$ precede the rectangles in $R_T^A$. In practice, this means that we can apply a simplified version of plane-sweep for $R_T^A \bowtie S_T^C$ which performs forward scans on $R_T^A$ for each rectangle in $S_T^C$, as the $r.x_l > s.x_l$ holds by definition. This modification will save on unnecessary comparisons between

**Algorithm 4.3** Reduced plane-sweep mini-join

**Require**: classes of rectangles $R_T$ and $S_T$

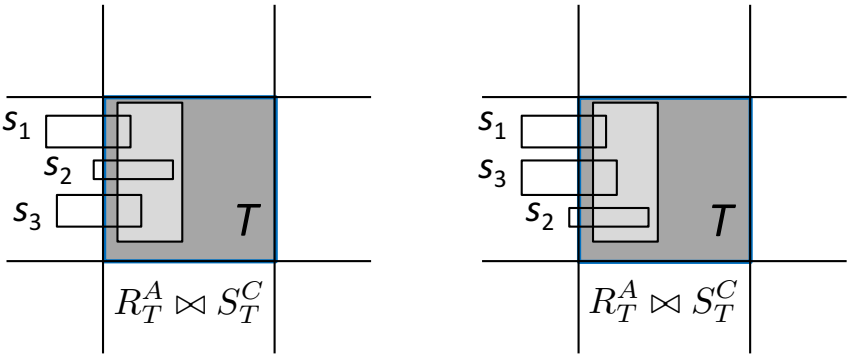
1: **sort** $R_T$ by $r.x_l$ ▷ `if not already sorted`
2: **while** $S_T$ not depleted **do**
3: $r' \leftarrow r$
4: **while** $r' \neq$ null **and** $s.x_u \geq r'.x_l$ **do**
5: **if** $r'.y_l \leq s.y_l \leq r'.y_u$ **or** $s.y_l \leq r'.y_l \leq s.y_u$ **then**
6: **output** $(r', s)$ ▷ `update result`
7: **end if**
8: $r' \leftarrow$ next rectangle in $R_T$ ▷ `scan forward`
9: **end while**
10: **end while**

$r \in R_T^A$ and $s \in S_T^C$ objects and further will allow us to avoid sorting the $S_T^C$ class. Algorithm 4.3 presents this reduced version of the plane-sweep mini-join. We can apply the same principle also for $R_T^A \bowtie S_T^C$, $R_T^C \bowtie S_T^A$, $R_T^A \bowtie S_T^D$, $R_T^D \bowtie S_T^A$, $R_T^B \bowtie S_T^C$ and $R_T^C \bowtie S_T^B$. Overall, we do not need to sort the $R_T^C$, $R_T^D$, $S_T^C$, $S_T^D$ classes.

Besides the plane-sweep process, we can also avoid unnecessary comparisons when considering the second dimension (i.e., $y$). The idea is similar to Lemmas 4.3 and 4.4. Take as example, the $R_T^A \bowtie S_T^B$ mini-join. For every pair $(r, s)$ of intersecting object rectangles in $x$, determined by plane-sweep, the $s.y_l < r.y_l$ condition holds by definition. Hence, to determine whether $r$, $s$ intersect also in dimension $y$ we only need to check the $r.y_l < s.y_u$ condition. The same principle can be used for the $R_T^A \bowtie S_T^D$, $R_T^B \bowtie S_T^A$, $R_T^D \bowtie S_T^A$, $R_T^B \bowtie S_T^C$ and $R_T^C \bowtie S_T^B$ mini-joins. The following lemma summarizes this optimization for every object rectangle $r$ in classes $R_T^B$, $R_T^D$ with $s$ in classes $S_T^A$ and $S_T^C$; the other case is symmetric.

**Lemma 4.6.** *If an object rectangle $r$ in tile $T$ starts before the tile in the $y$ dimension, i.e., $T.y_l > r.y_l$, then $r$ intersects all $s$ rectangles in $T$ that start after $T.y_l$ with $r.y_u > s.y_l$.*

**Avoid redundant comparisons**. To illustrate this optimization, consider the $R_T^A \bowtie S_T^C$ mini-join and the rectangles in Figure 4.10(a). As already discussed, the reduced plane-sweep approach (Algorithm 4.3) does not sort the $S_T^C$ class, to evaluate this mini-join; by definition, the start $s.x_l$ for their contents precede the $r.x_l$ start of all rectangles in $R_T^A$. Hence, assume that the $S_T^C$ rectangles are examined in the $s_1$, $s_2$, $s_3$ order. Rectangles $s_1$ and $r$ intersect in the $x$ dimension as $s_1.x_u > r.x_l$ holds and so,




(a) $S_T^C$ unsorted (b) $S_T^C$ sorted

Figure 4.10: Avoiding redundant comparisons on $R_T^A \bowtie S_T^C$; $R$ rectangle filled in light gray color.

they are next compared in the $y$ dimension. Algorithm 4.3 will similarly determine that both $s_2$ and $s_3$ also intersect $r$ in $x$ by checking $s_2.x_u > r.x_l$ and $s_3.x_u > r.x_l$, respectively. However, since $s_2.x_u > s_1.x_u$ and $s_3.x_u > s_1.x_u$ hold, the $s_1.x_u > r.x_l$ check for $s_1$ automatically implies that $s_2.x_u > r.x_l$ and $s_3.x_u > r.x_l$ also hold. In other words, we conducted two extra comparisons to determine the intersecting $(r, s_2)$ and $(r, s_3)$ pairs. To avoid such redundant comparisons, we can sort $S_T^C$ by $s.x_u$ which will essentially allow us to determine intersecting rectangles in batches. Figure 4.10(b) illustrates this idea; all three intersecting pairs can be determined after checking only $s_1$ against $r$. Algorithm 4.4 modifies Algorithm 4.3 accordingly. After identifying the first rectangle $s$ that intersects current $r'$ in the $x$ dimension (Line 5), the algorithm pairs $r'$ with every rectangle $s'$ that follows $s$ in $S_T$ (Line 6).

#### 4.2.5.2 Join Strategies

We last discuss different join strategies depending on the (pre)-existence of two-layer partitioning in the input. Following the classification in [22], we consider three settings.

##### 4.2.5.2.1 Both Inputs Indexed

Under this setting, a two-layer partitioning already exists on each input dataset $R$, $S$ to answer other types of spatial queries, e.g., range queries. We distinguish between two cases with respect to the granularity of the pre-existing grids. If the two grids are identical, we can directly apply the mini-joins approach and its optimizations in

**Algorithm 4.4** Reduced plane-sweep mini-join with batch outputting

**Require**: classes of rectangles $R_T$ and $S_T$

1: **sort** $R_T$ by $r.x_l$ ▷ `if not already sorted`
2: **sort** $S_T$ by $r.x_u$ ▷ `if not already sorted`
3: **while** $S_T$ not depleted **do**
4: $r' \leftarrow r$
5: **while** $r' \neq$ null **and** $s.x_u \geq r'.x_l$ **do**
6: **for** each rectangle $s'$ after $s$ in $S_T$ **do**
7: **if** $r'.y_l \leq s'.y_l \leq r'.y_u$ **or** $s'.y_l \leq r'.y_l \leq s'.y_u$ **then**
8: **output** $(r', s')$ ▷ `update result`
9: **end if**
10: $r' \leftarrow$ next rectangle in $R_T$ ▷ `scan forward`
11: **end for**
12: **end while**
13: **end while**

Section 4.2.5.1.

If the pre-existing grids have different granularities, then the mini-joins approach is not directly applicable. A straightforward solution is to re-index one of the input datasets, e.g. $R$, by creating a temporary two-layer partitioning with a grid granularity that matches the grid on $S$, similar to [165].

In this context, a key question that naturally arises is which input we should re-index. Typically, we could select the dataset with either the smallest cardinality or the smallest average object extent, because such a decision is expected to incur the lowest online indexing cost. We elaborate on this decision in our experimental analysis.

As an alternative, we also devise a new solution which completely eliminates the need to decide which input should be re-indexed and also significantly reduces the online indexing costs. For this purpose, the granularity of the pre-existing grids must be in a power of 2. If so, we can always define an online transformation of the finer grid to the coarser, by means of standard window range queries. Figure 4.11 exemplifies this transformation when $R$ is partitioned by a $2 \times 2$ grid (colored in blue), and $S$ by a $8 \times 8$ grid (in black). After overlaying the two grids, we observe that every tile $T$ in the $R$ grid overlaps a collection of exactly 16 adjacent tiles from the $S$ grid. Therefore, to re-partition $S$ to the $2 \times 2$ grid of $R$, it suffices to compute the window

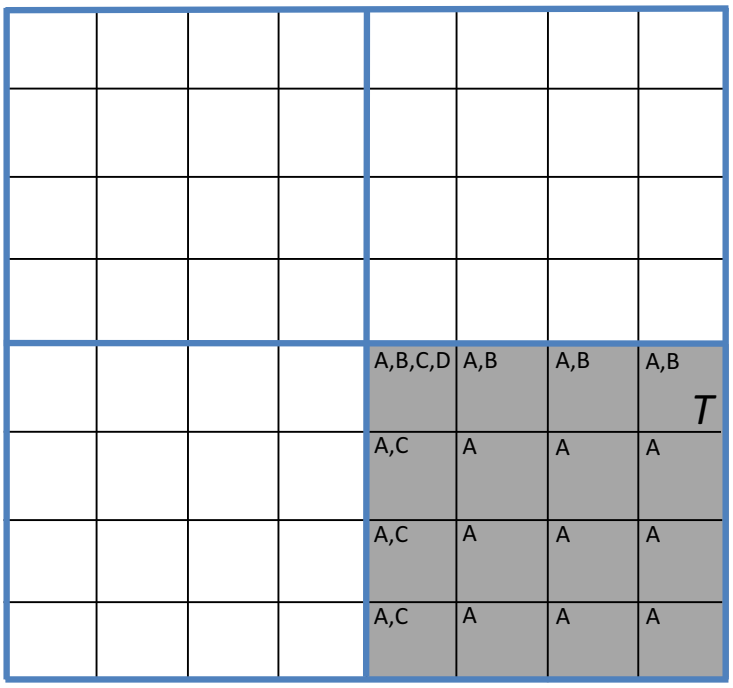



Figure 4.11: Online grid transformation; $R$ partitioned by a $2 \times 2$ grid, highlighted in blue, $S$ partitioned a $8 \times 8$ grid, in black.

queries defined by all tiles in the $R$ grid; the key idea is to determine the contents of the new $S_T^A$, $S_T^B$, $S_T^C$ and $S_T^D$ classes based on the results of every $T$ window query. For this purpose, we can utilize the original two-layer partitioning on $S$. Specifically, consider the shaded tile $T$ from the $R$ grid in Figure 4.11. Similar to Figure 4.6, we mark the relevant classes for each tile from the original $S$ grid that overlaps with the window query based on $T$, as discussed in Section 4.2.2. We can now construct $S_T^A$ by unifying the objects contained in the $A$ class of all overlapping tiles. For $S_T^B$, we consider only the contents of the $B$ class inside the top border overlapping tiles, while for $S_T^C$, the left border overlapping ones. Finally, $S_T^D$ is identical to the $D$ class inside the top left corner overlapping tile. The above process can be generalized for any two pre-existing grids of granularity $n \times n$ and $m \times m$, where $n > m$ and $n$, $m$ are powers of 2. Every tile in the second, coarser grid defines a window query that overlaps with exactly $(n/m)^2$ tiles from the first, finer one.

We developed two variants for the above transformation. The first constructs a temporary two-layer partitioning on input $S$ by matelializing the contents of each new $S_T^A$, $S_T^B$, $S_T^C$ and $S_T^D$ classes. After this, we can directly utilize the mini-joins approach in Section 4.2.5.1 to compute the $R \bowtie S$ spatial join. In contrast, the second variant never actually constructs this new two-layer scheme on input $S$. Instead, we

adjust the joining process in Section 4.2.5.1 to determine on-the-fly which tiles from the original grid on $S$ should be used in the mini-joins. Specifically, the $R_T^A \bowtie S_T^A$ is further decomposed to $(n/m)^2$ mini-joins, i.e., $R_T^A \bowtie S_T^A = \bigcup_i R_T^A \bowtie S_{T_i}^A$, where $T_i$ denotes one of the $(n/m)^2$ tiles from the original $S$ grid overlapping with tile $T$ from the $R$ grid. In the same spirit, $R_T^A \bowtie S_T^B$, $R_T^A \bowtie S_T^B$, $R_T^B \bowtie S_T^C$ and $R_T^C \bowtie S_T^B$ are decomposed into $(n/m)$ mini-joins, while $R_T^B \bowtie S_T^A$, $R_T^C \bowtie S_T^A$ and $R_T^D \bowtie S_T^A$ into $(n/m)^2$. In Section 4.2.6, we compare the two variants and break down their total execution time.

#### 4.2.5.2.2 One Input Indexed

Under this setting, a two-layer partitioning already exists only for one of the input datasets; without loss of generality, assume for $S$. In this case, there exist two alternative approaches for computing the $R \bowtie S$ spatial join. The first is to construct a temporary two-layer partitioning on $R$ with a grid of identical granularity to the grid on $S$, such that we can directly apply the joining process in Section 4.2.5.1. Despite its simplicity, this approach may exhibit high total execution times because of the online indexing cost.

Alternatively, we can adopt an index-based join approach. According to this, we scan the contents of the $R$ input and probe the index on $S$. Specifically, we issue a window range query for each object rectangle $r$ in $R$ which is evaluated using the two-layer partitioning on $S$. To further enhance the performance of this approach, we can examine the objects in $R$ according to their position in space, instead in a random order, e.g., by first partitioning them online with a grid or using a space filling curve. This way, window queries for objects in $R$ that overlap neighboring parts of the $S$ grid are evaluated in nearby timestamps, improving the cache locality.

#### 4.2.5.2.3 No Input Indexed

Under this setting, none of the input datasets $R$, $S$ is indexed by our two-layer partitioning. In this case, we can partition both inputs under identical grids to construct two temporary two-layer partitioning schemes and then directly apply the mini-joins approach from Section 4.2.5.1.

Since both inputs are indexed online, we can further enhance the join process by adopting a specialized storage optimization. Note that such an optimization cannot be

utilized for the settings discussed in the previous sections, as at least one of the two-layer schemes used for the join, pre-exists in order to evaluate other types of queries. Essentially, if we enforce this optimization, the resulting two-layer partitioning will no longer be able to fulfil its original purpose.

Conventionally, each rectangle $r$ is stored as a quintuple $\langle id, r.x_l, r.x_u, r.y_l, r.y_u \rangle$. Assuming $x$ as the sweeping dimension (the other case is symmetric), $r.y_l$ is never needed for classes $B$ and $D$ when we check whether two rectangles intersect also in the $y$ dimension, i.e., in Lines 6 and 14 of Algorithm 4.2, Line 5 of Algorithm 4.3 and in Line 7 of Algorithm 4.4. This is because all contained rectangles start before the start of the tile in $y$. In fact, for class $D$, we do not need $r.x_l$ either, since $D$ is only joined to an $A$ class from the other input $S$ and its contents always precede those $S$ rectangles in both dimensions. Hence, a temporary two-layer partitioning stores all information for rectangles in classes $A$, $C$ but $\langle id, r.x_l, r.x_u, r.y_u \rangle$ for $B$ and $\langle id, r.x_u, r.y_u \rangle$ for $C$, which reduces the online partitioning costs.

#### 4.2.5.3 Extension to other SOPs

The two-layer partitioning join in Section 4.2.5.1 is directly applicable to any SOP, provided that identical partitions of the space (defined either offline or online) are considered for both inputs. Similarly, the index-based join approach can be applied when one of the inputs is indexed any SOP, enhanced with our two-layer partitioning. Lastly, when both inputs are indexed by the same SOP but under a different set of partitions, the re-partitioning approach in Section 4.2.5.2.1 can be applied when there exists an alignment among the partitions; e.g., in quad-trees, for each a quadrant (of a coarse granularity) in one input that entirely covers a set of finer quadrants from the other.

### 4.2.6 Experimental Evaluation

In this section we present our experimental analysis. We first describe our setup and then present our experiments, which evaluate the effectiveness of our secondary partitioning approach by comparing a SOP index that employs it with a number of SOP and DOP indices.

Table 4.6: Real-world datasets used in the experiments

| dataset | type | cardinality | avg. relative [%] | |
|---|---|---|---|---|
| | | | $x$-extent | $y$-extent |
| ROADS | linestrings | 19M | 0.007 | 0.013 |
| EDGES | polygons | 69M | 0.003 | 0.005 |
| ZCTA5 | polygons | 33K | 1.7 | 2.052 |
| TIGER | mixed | 97M | 0.004 | 0.008 |

Table 4.7: Synthetic datasets (MBRs) used in the experiments

| parameter | values | default |
|---|---|---|
| cardinality | 1M, 5M, 10M, 50M, 100M | 10M |
| area | $10^{-14}$, $10^{-12}$, $10^{-10}$, $10^{-8}$, $10^{-6}$ | $10^{-10}$ |
| distribution | Uniform or Zipfian ($a = 1$) | — |

#### 4.2.6.1 Setup

Our analysis was conducted on a machine with 384 GBs of RAM and a dual Intel(R) Xeon(R) CPU E5-2630 v4 clocked at 2.20GHz running CentOS Linux 7.6.1810. All methods were implemented in C++, compiled using `gcc` (v4.8.5) with flags `-O3`, `-mavx` and `-march=native`. For our parallel processing tests, we used OpenMP and activated hyper-threading, allowing us to run up to 40 threads.

**Datasets.** We experimented with publicly available Tiger 2015 datasets [17], summarized in Table 4.6. The third dataset resulted by merging all polygon and linestring Tiger 2015 objects, excluding zips, counties and states. The objects in each dataset were normalized so that the coordinates in each dimension take values inside $[0, 1]$. The last two columns of the table are the relative (over the entire space) average length for every object's MBR at each axis. In order to test the robustness of our index, we also experimented with synthetically generated datasets with rectangles of uniform and zipfian spatial distribution. Table 4.7 shows the parameters used in data generation. The coordinates in each dimension take values inside $[0, 1]$ and all generated rectangles in a dataset have the same area. The width to height ratio of each rectangle was generated randomly in the range $[0.25, 4]$ in order to avoid unnaturally narrow rectangles.

**Methods**. We implemented our secondary partitioning approach as part of a main-

Table 4.8: Compared methods and their throughput (window queries)

| **type** | **index** | **throughput [queries/sec]** | |
|---|---|---|---|
| | | ROADS | EDGES |
| SOP | 2-layer$^+$ | 42856 | 14803 |
| | 2-layer | 36730 | 12942 |
| | 1-layer | 15068 | 5327 |
| | quad-tree | 13420 | 4564 |
| | quad-tree, 2-layer | 19354 | 6755 |
| DOP | R-tree | 10359 | 2935 |
| | R*-tree | 8886 | 2534 |
| | BLOCK | <1 | <1 |
| | MXCIF quad-tree | 8 | 2 |

memory regular grid spatial index. We designed two variants of the index. In the first variant, termed 2-layer, for each tile $T$ of the grid, we divide the (MBR, id) pairs assigned to $T$ into four secondary partitions ($T^A$, $T^B$, $T^C$, $T^D$), such that there is no particular order of the contents of each table (i.e., as in a heap file). This organization has low space requirements and supports updates efficiently as the MBRs of new objects are simply appended to the tables of the tiles. This variant discussed in Section 4.2.1.

In the second variant, termed 2-layer$^+$, each secondary partition $T^X$ is further divided into decomposed tables, as discussed in Section 4.2.2.3. 2-layer$^+$ takes advantage of the sorted decomposed tables to reduce the information that has to be accessed and the number of comparisons.

We considered both SOP and DOP competitors to our 2-layer and 2-layer$^+$, summarized in Table 4.8. First regarding SOPs, the 1-layer index is an in-memory grid with identical primary partitioning as our 2-layer, but uses the reference point approach [21] to perform duplicate elimination. Comparing 1-layer to 2-layer and 2-layer$^+$ shows the benefit of our secondary partitioning scheme and the techniques we propose in Section 4.2.2 for duplicate avoidance and minimization of comparisons. The second SOP competitor is a quad-tree implementation, which assigns each object MBR to all quadrants it intersects. As soon as the contents of a quadrant exceed a predefined maximum *capacity* (set to 1000, after tuning), the quadrant is split to four; the rectangles are then re-distributed in the four generated children and

*replicated* if they span the division borders. In order to avoid extensive splitting of quad-tree nodes in the case of extremely skewed data, a *maximum tree depth* (=12) is set. The reference point approach [21] is also used for duplicate elimination. We also implemented a version of quad-tree that uses our approach instead of [21]. Regarding DOPs, we used two implementations of in-memory R-trees from the highly optimized Boost. Geometry library (boost.org)[4]; an STR-bulkloaded [109] (denoted for simplicity as R-tree) and an R*-tree [96]. Both trees have a fanout of 16 for inner and leaf nodes; this configuration is reported to perform the best (we also confirmed this by testing). The next DOP competitor is BLOCK; the implementation was kindly provided by the authors of [88]. Finally, we also implemented and tested the MXCIF quad-tree for non-point data [167], which does not replicate objects that span quadrants, but stores each object at the lowest-level quadrant which covers the object. All compared methods are listed in Table 4.8.

**Queries**. We experimented with both window and disk queries which apply on non-empty areas of the map (i.e., they always return results). We vary (1) their relative area as a percentage of the entire data space, inside the $\{0.01, 0.05, 0.1, 0.5, 1\}$ value range (default value $0.1\%$ of the area of the map) and (2) their selectivity as a percentage of the returned objects over the cardinality of the dataset, inside the $[0, 0.001]$, $(0.001, 0.01]$, $(0.01, 0.1]$, $(0.1, 1]$ and $(1, 100]$ intervals. Queries on synthetic data follow the same spatial distribution as the data.

#### 4.2.6.2 Filtering vs. Refinement

We first justify our decision to focus on and optimize the filtering step of range query evaluation, which in fact has been the primary target of previous works as well. We used our 2-layer index to execute both the filtering and refinement steps. We consider three variants of query evaluation depending on the way refinement is performed (see Section 4.2.3); filtering is identical in all three variants. Specifically, under Simple, all candidates identified by the filtering step are passed to the refinement step; RefAvoid employs Lemma 4.5 as an extra pre-refinement filter to reduce the number of candidates to be refined; last, RefAvoid$^+$ enhances RefAvoid by using our secondary partitioning to reduce the number of comparisons required for testing Lemma 4.5,

[4] Recent benchmarks [166] showed the superiority of Boost.Geometry R-tree implementations over the ones in libspatialindex.org

as discussed in Section 4.2.3.

Figure 4.12 breaks down the average execution time for 10000 window and disk queries; note that for disk queries RefAvoid$^+$ is not applicable. The pre-refinement filter is very effective; both RefAvoid and RefAvoid$^+$ significantly reduce the number of candidates to be refined by over 90%. To achieve this however, they apply extra comparisons using the MBRs; these comparisons are more expensive in the case of disk queries because they involve costly distance computations between the disk center and the corners of object MBRs. Observe that, when our secondary filtering technique is used, not only the queries run faster but also the filtering step (including the pre-refinement filter) is now the most expensive step, instead of refinement in case with Simple. Therefore in the subsequent experiments, we focus on the filtering step of spatial query processing.

#### 4.2.6.3 Indexing and Tuning

We next investigate the index building cost and the tuning of two-layer indexing. The first four plots of Figure 4.13 compare the indexing times and the sizes of the three grid-based indices on ROADS and EDGES datasets, while varying the granularity of the grid partitioning. Naturally, the indexing cost for all three indices grows as we increase the granularity of the grid. As expected, 1-layer and 2-layer have the same space requirements; regardless of employing secondary partitioning or not, both indices store exactly the same number of object MBRs (originals and replicas). Note that the index sizes do not grow too much with the grid granularity, which means that MBR replication is not excessive. In terms of indexing time, 2-layer is only slightly more expensive than 1-layer, as it needs to first determine the class for each rectangle and then store it accordingly. On the other hand, the indexing cost of 2-layer$^+$ is higher than both 1-layer and 2-layer indices, because 2-layer$^+$ essentially stores a second (decomposed) copy of the rectangles inside every tile. The construction costs for the two quadtrees (not shown) are 7s and 28.2s, respectively, and their sizes are similar to those of the corresponding 1-layer indices. The sizes of the packed R-trees (not shown) are about the same as the sizes of the corresponding 1-layer (and 2-layer) indices, indicating that the replication ratio of our indices is low. In addition, the bulk loading costs of the R-trees are 5.2s and 19.5s for the two datasets, respectively, i.e., about 20% lower compared to the construction cost of 2-layer$^+$.

The last two plots of Figure 4.13 compare the window query throughputs of

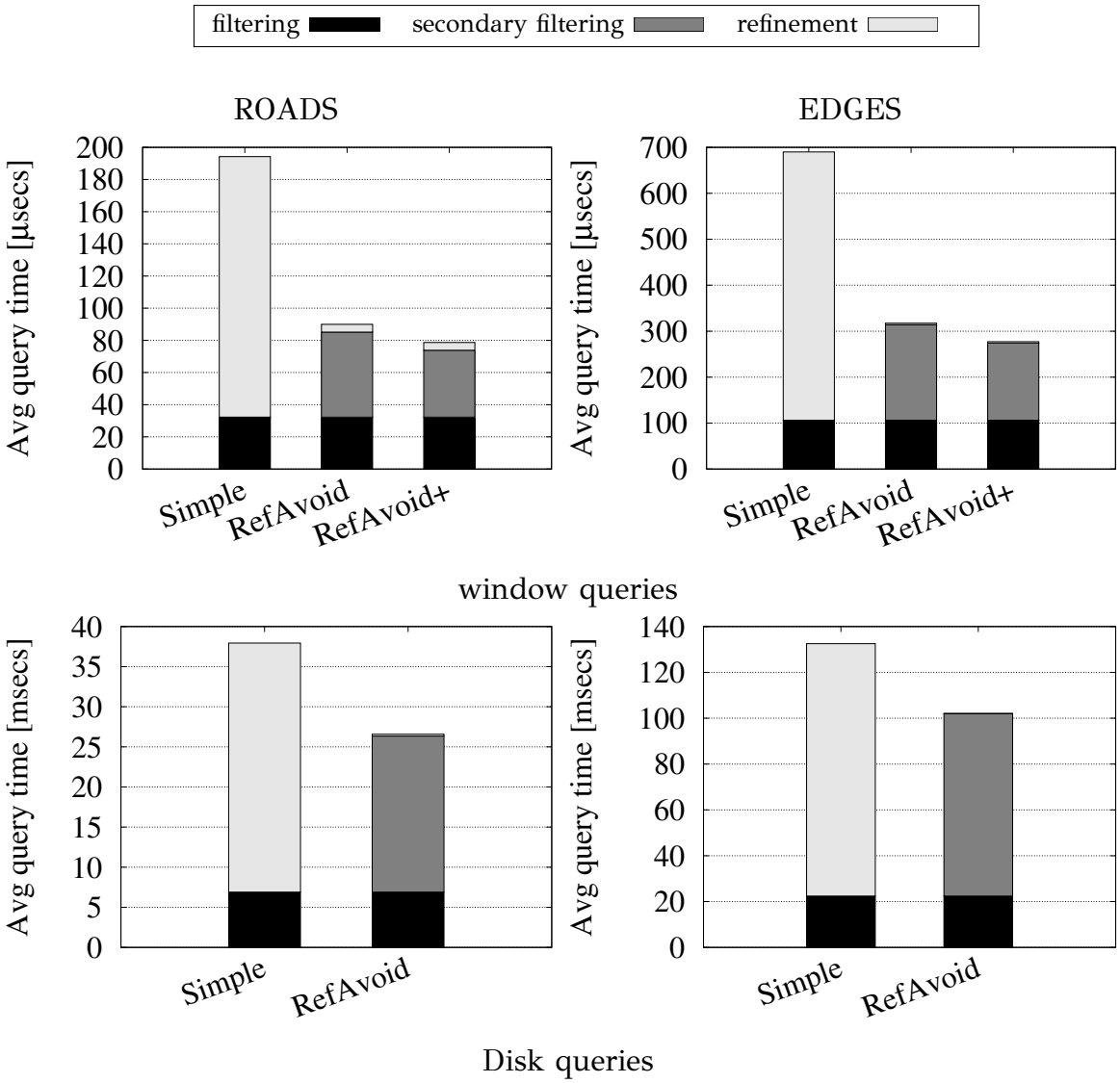



Figure 4.12: Time breakdown in two-layer indexing

1-layer, 2-layer, and 2-layer$^+$ for different grid granularities. The three methods achieve their best throughputs when several thousands of partitions per dimension are used. Under this configuration, the number of tiles is not excessive and the indices do not have a large overhead in accessing and managing tiles. A key observation is that employing our secondary partitioning significantly enhances query processing; 2-layer and 2-layer$^+$ always outperform 1-layer by a wide margin (2x–3x). It is worth noting that 1-layer uses the comparisons reduction optimization described in Section 4.2.2.2, meaning that the performance gap is due to our secondary partitioning and the storage decomposition (by 2-layer$^+$). Specifically, our approach outperforms the state-of-the-art reference point method for result deduplication [21] used by 1-layer by a factor of at least 2. For a wide range of granularities (i.e., 1000

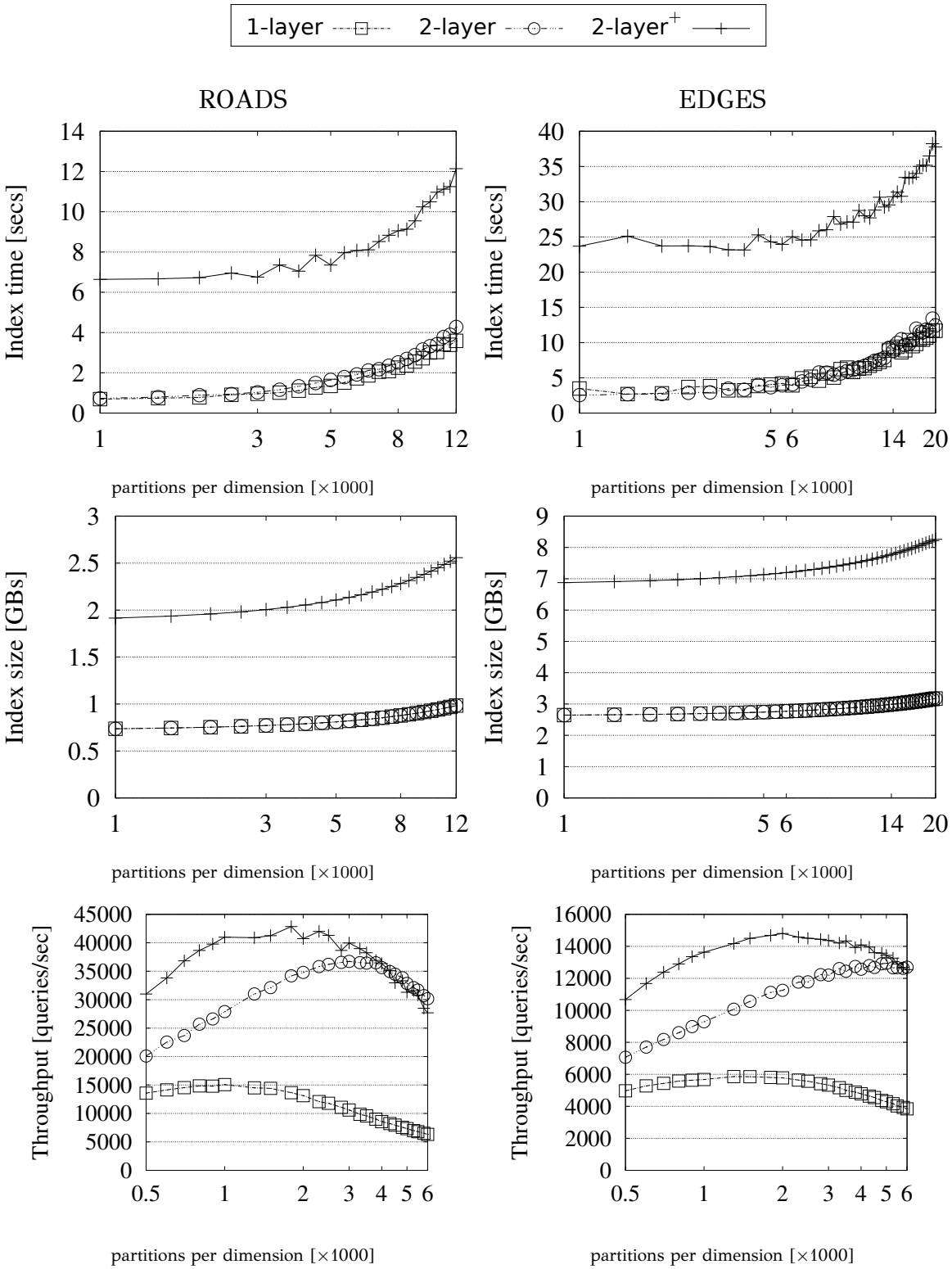



Figure 4.13: Building and tuning grid-based indices (window queries)

Table 4.9: Best granularities (partitions per dimension); in parenthesis, the power of 2 used for the transformation-based methods in Figure 13.

| index | ROADS | EDGES | ZTCA5 | TIGER | Uniform | Zipfian |
|---|---|---|---|---|---|---|
| 1-layer | 1000 | 3000 | not used | 1000 | 500 | 3000 |
| 2-layer | 3000 ($2^{11}$) | 5000 ($2^{12}$) | 400 ($2^{8}$) | 5300 | 450 | 3000 |
| 2-layer$^{+}$ | 1800 | 2000 | not used | 2300 | 450 | 3000 |

to 10000 partitions per dimension), the throughput of all three methods does not change significantly meaning that finding the best granularity is not crucial to query performance. The fastest index is 2-layer$^{+}$ as it trades the extra used space for better query performance We observed similar trends on the TIGER and on the synthetic datasets (not shown due to lack of space). For the rest of our analysis, we used the best granularity for 1-layer, 2-layer and 2-layer$^{+}$, see Table 4.9.

#### 4.2.6.4 Query and Update Performance

We now compare all indices in terms of query throughput (window and disk queries), evaluate batch and parallel query processing, and finally measure their update costs.

**Window queries.** First, we report in Table 4.8 the throughput (queries/sec) achieved by each index for 10K window queries (of average relative area 0.1% of the map) on ROADS and EDGES. 2-layer and 2-layer$^{+}$ outperform the competition by a wide margin. R-tree is the most efficient DOP competitor, outperforming R*-tree (from the same library). BLOCK takes seconds to evaluate range queries on our datasets, which can be attributed to the fact that it is implemented for 3D objects. Similar, MXCIF quad-tree is orders of magnitude slower than the R-tree. Under these, in the rest of the experiments we only include 1-layer, R-tree and quad-tree indices as the key competitors to our 2-layer and 2-layer$^{+}$.

The first two columns of Figure 4.14 show the throughput of the five competitors for window queries of varying relative area and selectivity on the three real datasets. For the experiments of the second column, we collected the runtimes of all queries (regardless of their areas) and averaged them after grouping them by selectivity. Naturally, query processing is negatively affected by both factors. As the window extent increases or the query becomes less selective, a larger number of objects overlaps the query area,rendering the range queries more expensive. We also observe that 2-layer

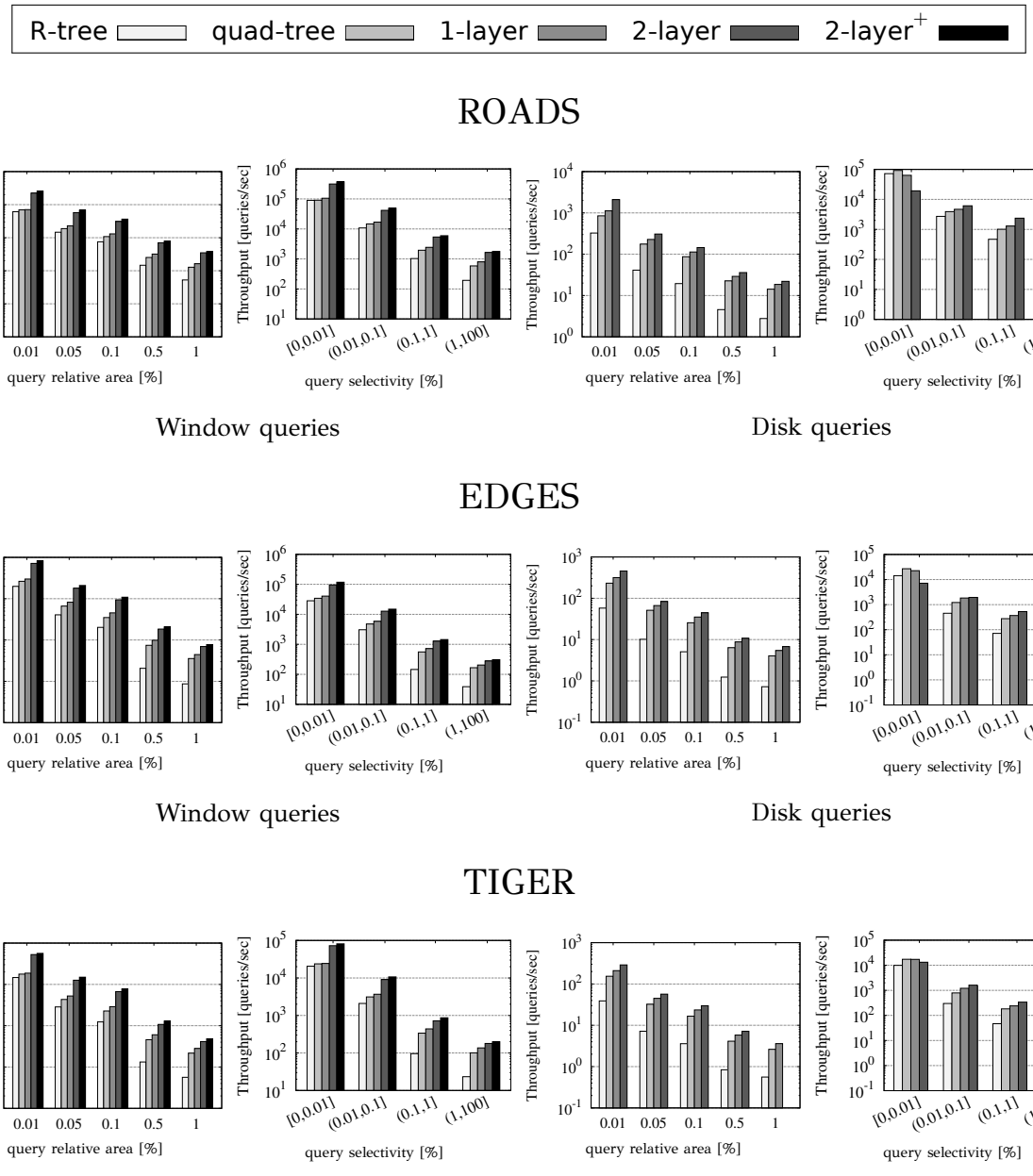



Figure 4.14: Query processing: real data

and 2-layer$^+$ are consistently much faster than the competition on all datasets and query areas; in addition, the relative difference between 2-layer and 2-layer$^+$ is stable. For each query, 2-layer and 2-layer$^+$ access the relevant partitions very fast (without the need of traversing a hierarchical index) and manage to drastically reduce the required number of computations. Figure 4.15 compares all methods for window queries on the synthetic datasets. In these experiments, we additionally vary the database size and the areas of the data objects. In terms of query throughput w.r.t. query area and selectivity, the trends are similar as those for the real data. In addition, the data cardinality does not affect the relative performance of the methods. Finally, we observe that 2-layer and 2-layer$^+$ are more robust to the area of the data objects compared to the competition. As the area grows, the replication to tiles increases and 1-layer and the quad-tree have to compute and eliminate more duplicate results. On the other hand, 2-layer and 2-layer$^+$, with the help of our secondary partitioning, avoid the generation and elimination of duplicate results.

**Disk range queries**. We now turn our focus to disk range queries. For the interest of space, we only report results on the real data in the last two columns of plots in Figure 4.14. 2-layer$^+$ is not included in the comparison, because its storage decomposition does not give any benefit compared to 2-layer, as all coordinates are needed in distance computations. Since, for disk queries on 1-layer and quad-tree, we cannot use the reference point technique to eliminate duplicate results (and duplicate elimination using hashing is too expensive), we implemented disk queries on them as follows. We executed a window query using the MBR of the query range and eliminated any duplicates intersecting the window. For all tiles/quadrants inside the disk range, we just reported all window query results there as disk query results. For all other tiles and quadrants we performed distance tests before confirming and reporting the results. The plots show once again the superiority of the 2-layer index.

**Batch and Parallel Query Processing.** Figure 4.16 compares the two approaches (queries-based and tiles-based), discussed in Section 4.2.4, for batch window query processing (10K queries or 1% relative area, per batch) on ROADS and EDGES.[5] A general observation from the plots is that tiles-based is superior to queries-based when the dataset is large (i.e., dense) and the queries are relatively large. In this case, the sizes of the dedicated tables for each class per tile are large and cache conscious

[5]Similar findings are observed for TIGER, but the results are omitted due to lack of space.

Table 4.10: Total update cost (sec)

| dataset | R-tree | quad-tree | 1-layer | 2-layer |
|---|---|---|---|---|
| ROADS | 5.34 | 0.76 | 0.059 | 0.068 |
| EDGES | 19.8 | 2.89 | 0.267 | 0.382 |
| TIGER | 33.91 | 4.63 | 0.459 | 0.634 |

tiles-based approach makes a difference. On the other hand, the overhead of finding and accumulating the subtasks per tile does not pay off when the number of queries on each tile is too small or when the tiles do not contain many rectangles. The advantage of tiles-based becomes more prominent in parallel query processing. Figure 4.17 shows the speedup of batch query evaluation on the two largest datasets (again, 10K queries per batch) as a function of the number of parallel threads. Note that tiles-based scales gracefully with the number of threads (up to about 25 threads, where it starts being affected by hyperthreading). On the other hand, queries-based scales poorly due to the numerous cache misses.

**Updates.** To confirm the superiority of grid indices in updates, we conducted an experiment using the real datasets, where we first constructed the index by loading 90% of the data in batch and then measuring the cost of incrementally inserting the last 10% of the data. Table 4.10 compares the total update costs of the competitor indices. R-tree is two orders of magnitude slower than the baseline 1-layer index and the cost of updates on 2-layer is only a bit higher compared to the update cost on 1-layer. Updates on quad-tree are also slower compared to 1-layer and 2-layer, due to the tree traversal.

#### 4.2.6.5 Comparison with GeoSpark

Finally, we compare our proposed 2-layer grid index with GeoSpark [24], one of the best-performing distributed spatial data management systems according to [20]. Our goal is to show that, for the scale of benchmarking data [17] that we and recent papers [20, 34, 35] use, in-memory indexing in a multi-core processing machine is superior to using a system designed for cluster computing. As our implementation is designed to run on a single machine, we run GeoSpark in client mode, meaning that the driver and Spark applications are both on the same machine. In addition, we used R-tree indexing in GeoSpark, which performs best in query evaluation. We

compared GeoSpark with 2-layer that uses a grid granularity of 1000x1000 and tested both single and multi-threaded versions for range queries. The experiments were conducted using the ROADS dataset on a machine with 64 GBs of RAM and a Intel(R) Core i7-4930K CPU clocked at 3.40GHz.[6] For each method, we average the cost of 100 (end-to-end) window queries, where the area of each query is 0.1% of the area of the map. Figure 4.18 shows that 2-layer always outperforms GeoSpark in terms of query performance by at least three orders of magnitude. These results are consistent with the findings of [20], where distributed spatial data management systems are shown to have a throughput of at most several hundred range queries per minute on data of similar scale. In order to compare the two methods in a multi-threaded scenario on equal terms, our approach evaluates the queries independently (i.e., not in batch). We observe the same trend as the number of cores increases.

#### 4.2.6.6 Spatial Join Performance

**Two-layer Partitioning Join.** For the spatial intersection join, we use only 2-layer, because 2-layer$^+$ is more complex. We first study the impact of our second layer of partitioning to the join computation. As discussed in Section 4.2.5.1, we assume that both input datasets are already indexed and that identical grids exist as the first layer of partitioning. We implemented the *mj, base* method on top of our 2-layer scheme, as our basic mini-joins solution which adopts the mini-joins breakdown from Section 4.2.5.1.1 and the plane-sweep join approach from Section 4.2.5.1.2. To demonstrate the effect of the comparison-saving optimizations from Section 4.2.5.1.2, we also developed the *mj, sans unecessary* and *mj, sans redundant* variants, which extend *mj, base*. Last, we implemented a mini-joins solution with all optimizations activated, denoted by *mj, all opts* and 1-layer, a PBSM baseline solution that solely employs the first layer of partitioning.

Figure 4.19 reports the breakdown of the total execution time while varying the number of partitions per dimension for the ROADS $\bowtie$ ZCTA5 and EDGES $\bowtie$ ZCTA5 queries. Note that we excluded the offline partitioning time but included the sorting time needed for adopting plane-sweep and saving on redundant comparisons.[7] The

[6] We do not own the platform where we ran the previous experiments, so we could not install GeoSpark on that machine.

[7] Note that 1-layer employs the plane-sweep approach in [18] for partition-to-partition joins, similar to our 2-layer based methods.

results clearly show the benefit of employing our second layer of partitioning and the mini-joins breakdown compared to 1-layer. We also observe that employing all comparison-saving optimizations can further accelerate the join computation; the *mj, all opts* always outperforms the rest tested methods.

**Both Inputs Indexed.** We then experiment with the three join strategies discussed in Section 4.2.5.2, starting with the setting where both input datasets are indexed by a 2-layer scheme. The granularity of each first layer grid is set according to our analysis in Section 4.2.6.3 as powers of 2 (see Table 4.9); i.e., the pre-existing 2-layer schemes are optimized for range queries. We consider again the ROADS $\bowtie$ ZCTA5 and EDGES $\bowtie$ ZCTA5 queries.

Figure 12 reports the time breakdown of the tested approaches; as a baseline, we also included a traditional R-tree join method [18]. We tested the straightforward approach of re-indexing one of the inputs using a temporary 2-layer so that both indices employ identical grids and being able to use the *mj, all opts* join method from the previous section. The figure shows the results for re-indexing ROADS and EDGES, while re-indexing ZCTA5 is omitted. As ZCTA5 contains significantly larger rectangles than ROADS and EDGES (see Table 4.6), indexing these rectangles under the very fine grids used by 2-layer for ROADS and EDGES incurs a high replication ratio and hence, high partitioning costs (over 300 secs). The figure also includes the two variants for the online transformation of the 2-layer scheme in ROADS and EDGES to the grid of ZCTA5, proposed in Section 4.2.5.2.1. The results clearly show the benefit the online transformation compared to re-indexing one of the inputs and in particular, the variant when the new 2-layer scheme is not materialized as such an approach completely eliminates the partitioning costs.

**One Input Indexed.** We consider once again the ROADS $\bowtie$ ZCTA5 and EDGES $\bowtie$ ZCTA5 queries when only one input is indexed by our 2-layer scheme. We tested three solutions in this context. The first is a classic index-based join; we denote this approach as *for-loop, probe*. The second approach *grid* $10 \times 10$*, probe* extends this idea by partitioning the unindexed input with a coarse grid (we used a $10 \times 10$ one) and then executing the window range queries per grid cell. Finally, we also employed the 'Both Inputs Indexed' approach by indexing the unindexed input online.

Figure 13 reports the breakdown of the execution time; we distinguish between two cases for each query, depending on which input is already indexed. Note that we

omit the 'Both Inputs Indexed' approach when ZCTA5 has to be indexed, similarly to previous section. We discuss the ROADS $\bowtie$ ZCTA5 query as the findings are the same for EDGES $\bowtie$ ZCTA5. When ZCTA5 is indexed, we observe that building online a temporary 2-layer scheme on ROADS with an identical grid to ZCTA5 is in fact the best solution. This is mainly because *mj*, *all opts* used to compute the join drastically reduces the joining time. In contrast, when ROADS is pre-indexed, the best approach is *grid* $10 \times 10$, *probe* which improves the cache locality when probing the index on ROADS. As ZCTA5 contains significantly fewer rectangles than ROAD, the cost of constructing such a grid is negligible compared to applying the same idea on ROADS when ZCTA5 is pre-indexed. To sum up, when one of the inputs is pre-indexed by our 2-layer scheme, an $R \bowtie S$ spatial join can be efficiently computed using the 'Both Inputs Indexed' strategy if the cost of indexing the second input is expected to be low (due to containing small rectangles or using a coarse first layer grid) or the grid-based probe approach otherwise.

**No Input Indexed.** Lastly, we consider the setting when none of the inputs is indexed. As discussed in Section 4.2.5.2.3, we directly construct in this case, two temporary 2-layer schemes under an identical first layer grid and then use *mj*, *all opts* to compute the join. Figure 4.22 shows the effect of further optimizing this approach with the space reduction optimization proposed at the end of Section 4.2.5.2.3, while varying the number of partitions per dimension. We denote this method as *mj*, *all opts* + *s-opt*. We observe that the space optimization enhances the join computation, especially when a very fine grid is used.

## 4.3 Conclusions

In this chapter, we investigated directions towards tuning a classic and popular partitioning-based spatial join algorithm, which is typically used for in-memory and parallel/distributed join evaluation. We experimented with varying the number and type of partitions and the sweeping axis choice in plane sweep. We also designed an efficient parallel version of the algorithm. Our experimental findings show that 1D partitioning performs better than 2D and that the correct sweeping axis choice does matter. In addition, we showed that the parallel version of the algorithm scales well with the number of threads.

We also presented a secondary partitioning approach that can be applied to SOP

indices, such as grids, and divides the MBRs within each spatial partition to four classes. Our approach reduces the number of comparisons during range query evaluation and avoids the generation (and elimination) of duplicate results. In addition, we proposed a secondary filtering technique for spatial range queries that avoids a refinement step for the majority of query results. Moreover, we investigated techniques for evaluating numerous range query requests in batch and in parallel. Finally, we show how our approach can also be used for duplicate result avoidance in spatial intersection joins that are evaluated using the state-of-the-art PBSM algorithm. For both range queries and joins, we show how redundant computations can also be avoided. Our experimental findings confirm the superiority of our approach compared to the state-of-the-art duplicate result elimination method [21]. We also show that a grid equipped with our method outperforms other indices (such as the quad-tree and R-tree) by up to one order of magnitude. The cost of spatial intersection joins is also reduced by a significant factor (about 50%) with the help of our secondary partitioning technique and the use of our optimized partition-to-partition join algorithms.

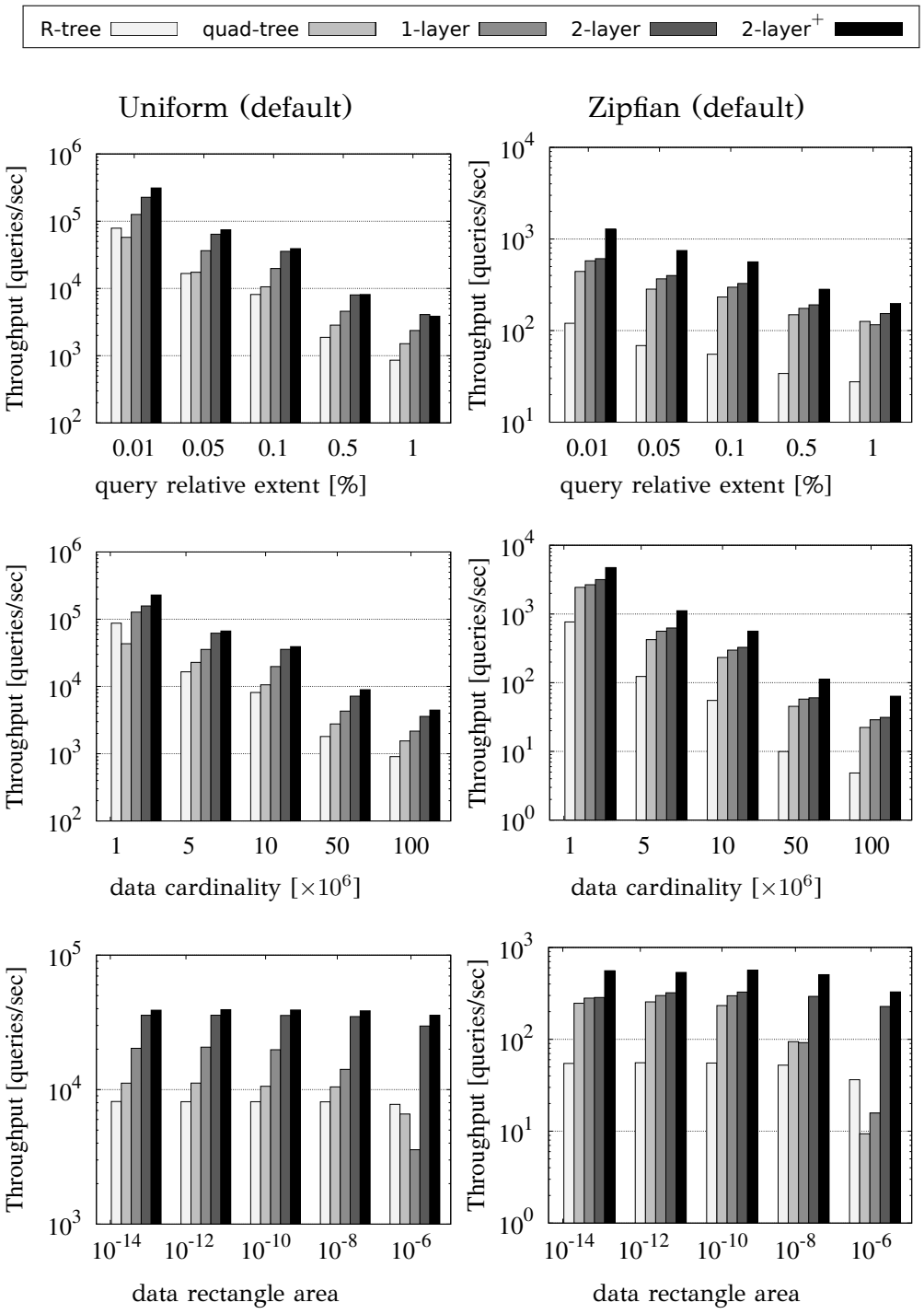



Figure 4.15: Query processing: synthetic data (window queries)

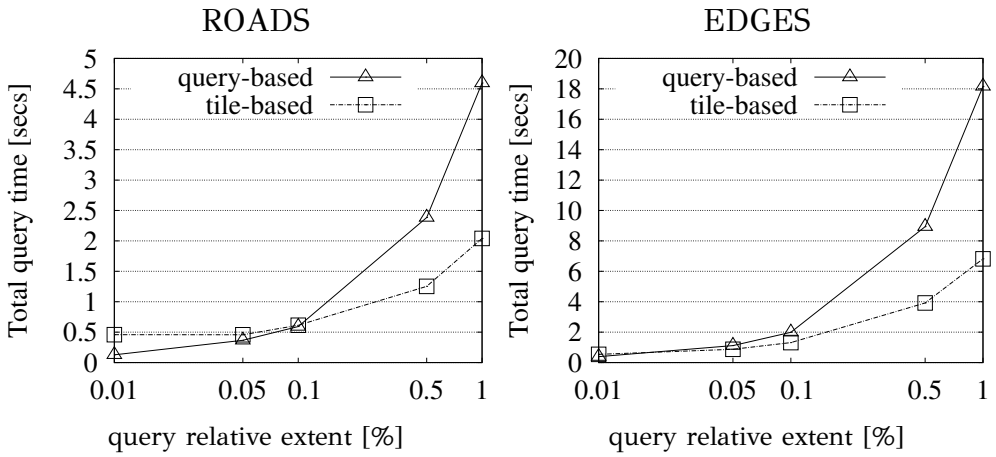



Figure 4.16: Batch query processing (window queries)

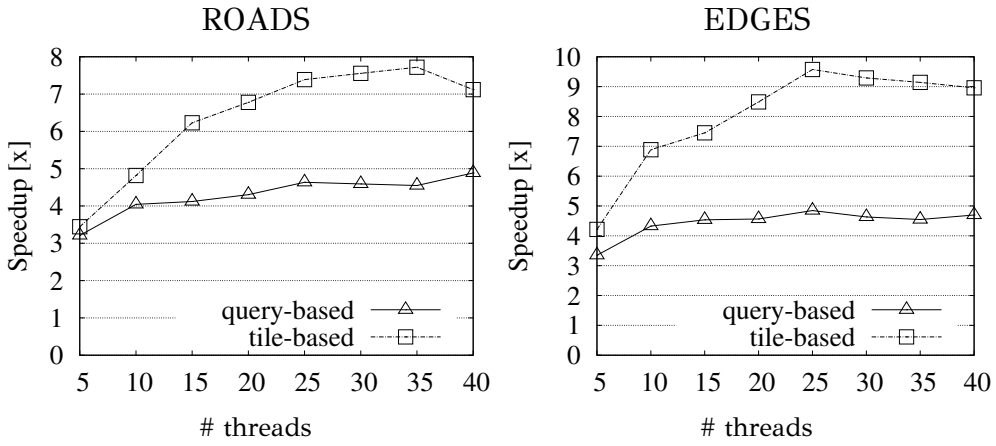



Figure 4.17: Batch query parallel processing (window queries)

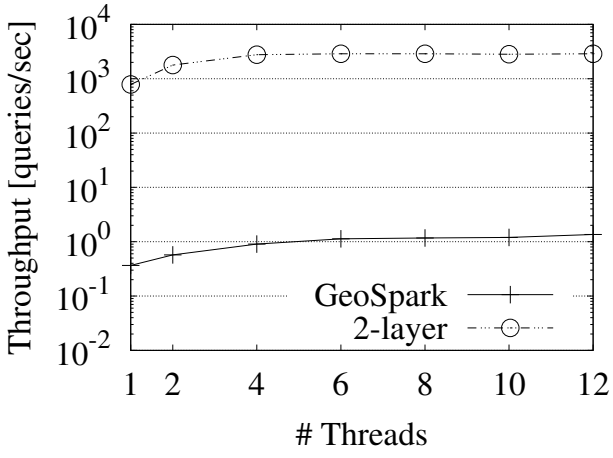



Figure 4.18: Window query performance comparison

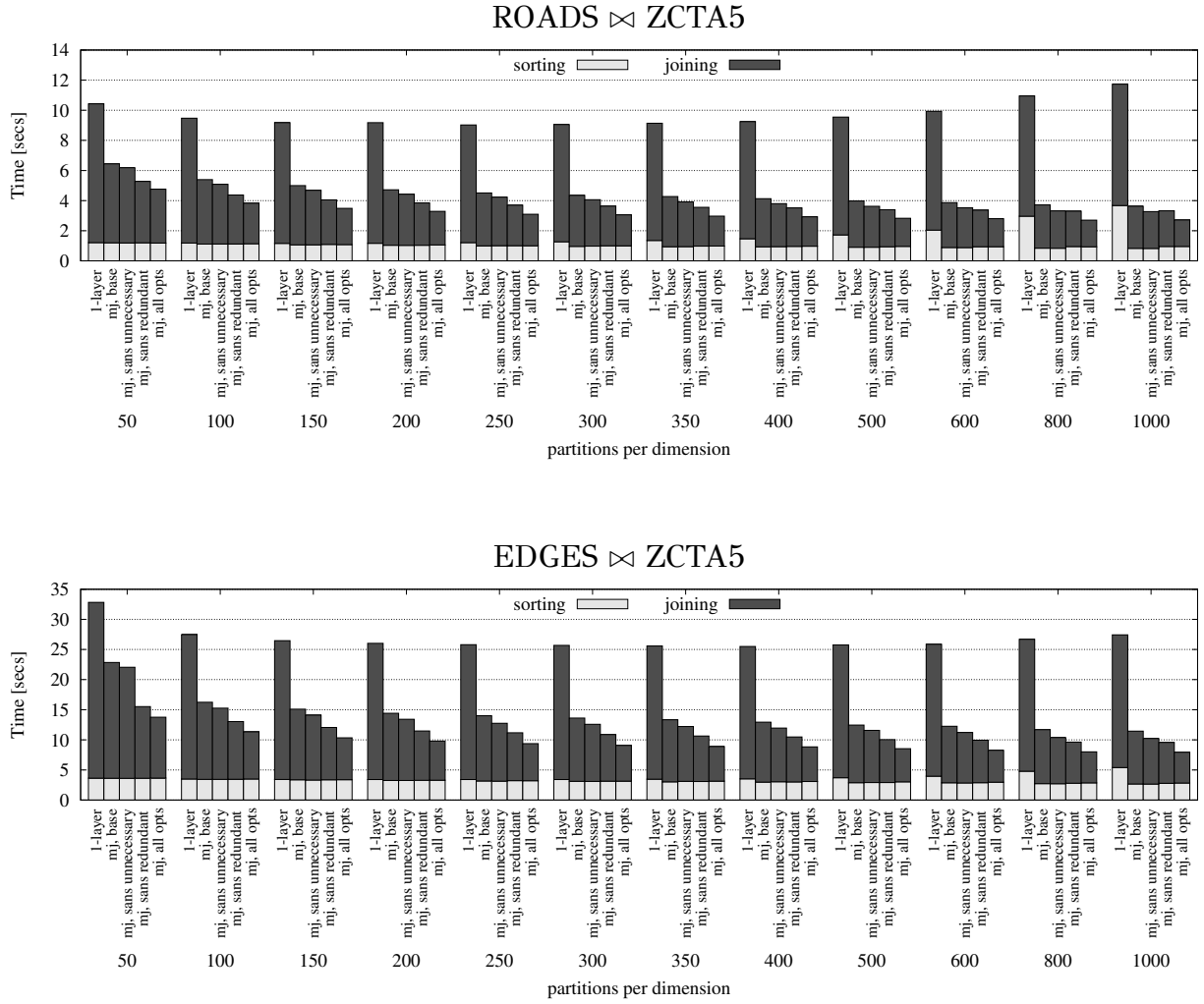



Figure 4.19: Two-layer partitioning join on real datasets: mini-joins breakdown and optimizations.

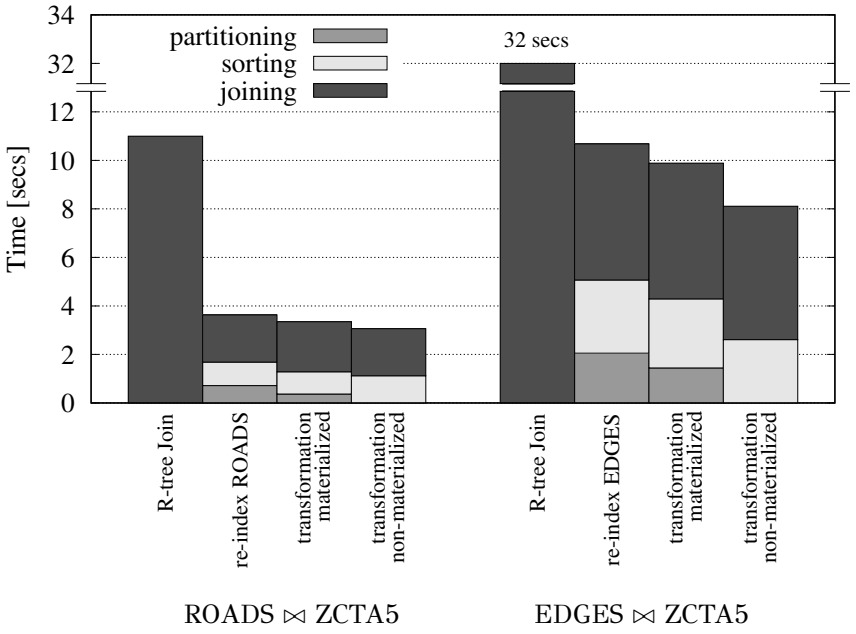



Figure 4.20: 'Both Inputs Indexed' setting on real datasets; re-indexing ZCTA5 omitted due to high online partitioning costs.

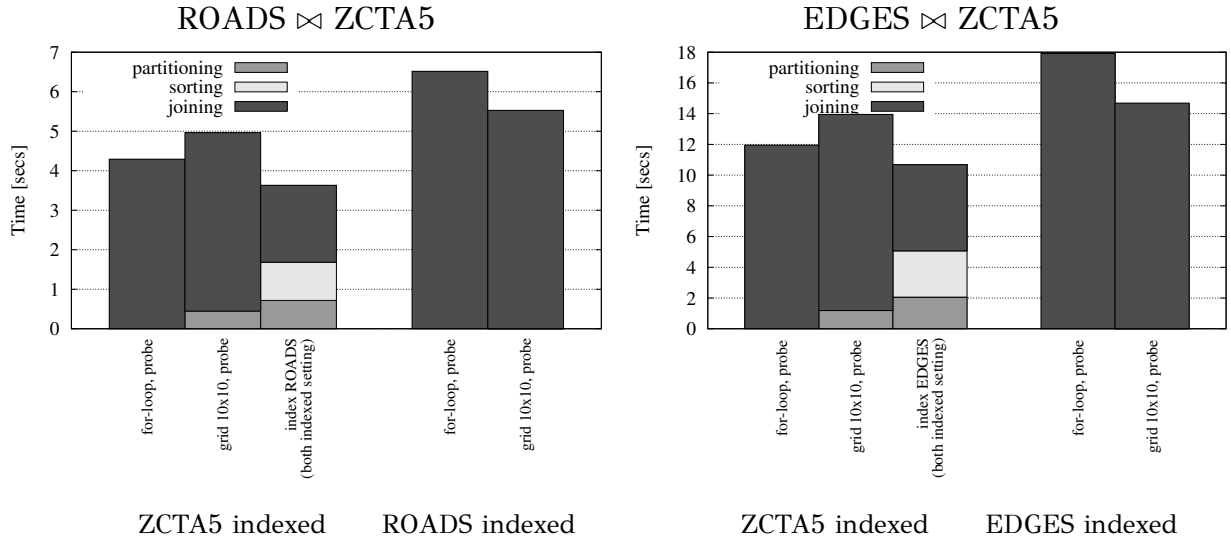



Figure 4.21: 'One Input Indexed' setting on real datasets; indexing ZCTA5 omitted due to high online partitioning costs.

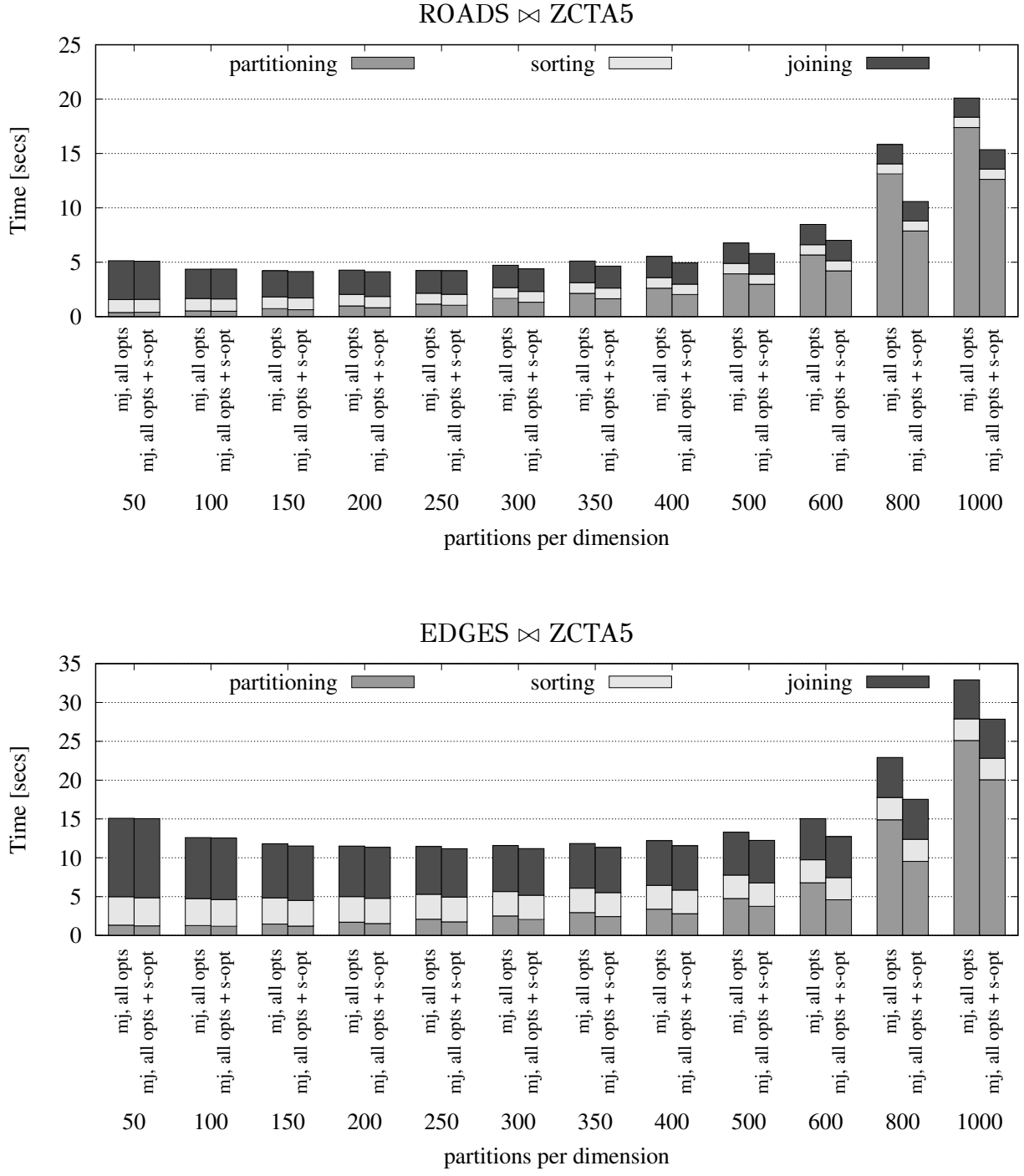



Figure 4.22: 'No Input Indexed' setting on real datasets

# Chapter 5

# $B^S$-tree: A Data-Parallel $B^+$-tree For Main Memory



As memories become cheaper and larger, main memory databases are becoming the standard approach for handling data, even in commodity machines. We propose $B^S$-tree, an in-memory implementation of the $B^+$-tree that adopts the structure of the disk-based index (i.e., a balanced, multiway tree), setting the node size to a memory block that can be processed fast and in parallel using SIMD instructions. Our proposal includes a parallelizable successor search operation that applies at each tree level to locate a key position or its successor. We propose a novel implementation for gaps (unused positions) within nodes by duplicating keys that facilitates fast operations (searches, insertions, deletions). We also present a compression mechanism, which can greatly decrease its size in memory without imposing a large performance overhead.

We compare our approach to existing main-memory indices and learned indices under different workloads of queries and updates and demonstrate its robustness and superiority compared to previous work.

**Outline** The $B^S$-tree is described in Section 5.1 and its updates and construction in Section 5.2. Section 5.3 describes $B^S$-tree compression. Implementation details and concurrency control are discussed in Section 5.4 and 5.5, respectively. Section 5.6 includes our experimental evaluation and we conclude in Section 5.7.

## 5.1 The $B^S$-tree

The $B^S$-tree that we propose follows the structure of the $B^+$-tree. Each internal node of the tree fits up to $N$ references to nodes at the lower level. We first present the data structure in Section 5.1.1. Then, Section 5.1.2 describes the implementation of the successor operator applied to each node for branching and key location during search and updates. Finally, Section 5.1.3 presents the algorithms for equality and range search.

### 5.1.1 The structure

$B^S$-tree follows the structure of the $B^+$-tree. Each internal node of the tree fits up to $N$ references to nodes at the lower level and up to $N-1$ keys. Leaf nodes contain rid-key pairs, where a record-id (rid) is the address (potentially on the disk) of the record that has the corresponding key value. We assume that keys are unique (if not, rid is replaced by a pointer to a block that keeps the rid's of all records having the corresponding search key). The storage of the rid's is decoupled from the storage of the keys, i.e., they are stored in two different arrays, such that the rid array is accessed only if necessary (i.e., only if the key is found and we need access to the corresponding record). Similarly, the storage of keys in an internal node is decoupled from the storage of node (memory) pointers, to facilitate fast search, as we explain later. Each leaf node hosts a node pointer to the next leaf; i.e., the leaves of the tree are chained based on the total order of the keys. Chaining is used for the efficient support of range queries as the results of such queries should be in consecutive tree leaves. For the $B^S$-tree nodes we use a value of $N$ that facilitates fast and parallel search, as we will explain later. For the efficient handling of updates, we allow gaps

(i.e., unused slots) in nodes, similarly to previous work [60, 64, 65].

Figure 5.1 shows an example of a B$^S$-tree, where each node holds up to $N-1=4$ keys. Each non-leaf node is shown as an array of $N$ node pointers (bottom) and $N-1$ keys (top) that work as separators. All keys in the subtree pointed by the $i$-th pointer are strictly smaller than the $i$-th key and greater than or equal to the $(i-1)$-th key (if $i>0$). Any unused key slots at the end of each node carry a special $\infty$ value, which is a MAXKEY value, larger than the maximum possible value in the key domain.

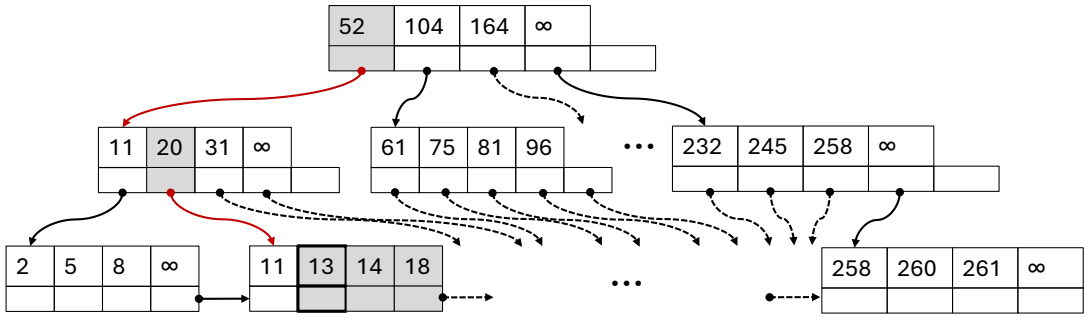



Figure 5.1: Example of B$^S$-tree

### 5.1.2 Search within a B$^S$-tree node

So far, besides the use of MAXKEY (i.e., $\infty$) values for unused slots at the end of each node, there are no differences between the B$^S$-tree and the B$^+$-tree. We now elaborate on the first major difference, which is the implementation of *branching* at each node of the B$^S$-tree, i.e., selecting the next node to visit. Traditionally, at each visited node, starting from the root, finding the smallest key which is strictly greater that the query key $k$ is done either by binary search or by linearly scanning the entries until we find the first key greater than $k$. Both these approaches incur significant CPU cost due to branch mispredictions.

We propose an efficient implementation of the *successor* operator applied to each node along the search path, which does not involve search decisions. We denote by $succ_>$ the operator that finds the smallest key position which is strictly greater that the query key $k$ (used in non-leaf nodes) and by $succ_\geq$ the finding of the smallest key position which is greater than or equal to $k$ (used in leaf nodes). For example, in Figure 5.1, $succ_{>124}(root)=2$, as the smallest key which is greater than 124 is at position 2. This means that if we are looking for key 124, we should follow the node pointer at position 2 of the root. In general, $succ_>$ is the most frequently applied

operation during search, as we use it on each non-leaf node along the path from the root to the (first) leaf node that includes the search result.

**Node operations** We first define and outline the operations that need to be applied at tree nodes during search:

- least greater than, denoted by $succ_{>}$: find the position of the smallest key which is strictly greater that the query key (for equality search) or the first qualifying key in the query key range such as $k_{low} < x \leq k_{high}$.

  This operation is applied to non-leaf nodes during search and possibly to the first visited leaf by a range query.

- least greater than or equal to, denoted by $succ_{\geq}$: find the position of the smallest key which is greater than or equal to the query key (for equality search) or the first qualifying key in the query key range. This operation is applied to the leaf node visited by the search.

Our approach exploits data parallelism (i.e., SIMD instructions) and does not include if-statements or while statements with an uncertain number of loops. The implementation of $succ_{>}$ is based on the following lemma:

**Lemma 5.1.** *Let $v$ be a node and $k$ be the search key. Then, $succ_{>k}(v) = |\{x : x \in v.keys \wedge k \geq x\}|$, where $|S|$ denotes cardinality of $S$.*

Lemma 5.1 basically states that the position in node $v$ that corresponds to the first key greater than $k$ equals the number of keys in $v$ which are smaller than or equal to $k$. The proof is straightforward given that (i) they keys in $v$ are sorted and (ii) the unused keys at the end of the node all have value MAXKEY, which is always greater than $k$. Based on Lemma 5.1, $succ_{>k}(v)$ can be implemented by each of the code Snippets 5.1 and 5.2 (the second one parallelized with SIMD instructions).

The code snippets do not include if-statements and do not incur branch mispredictions. CAPACITY is the (fixed) capacity of the node, so the number of iterations of the for-loop is hardwired; all these favoring data parallelism.

Similarly, the implementation of $succ_{\geq k}(v)$ is based on the following lemma:

**Lemma 5.2.** *Let $v$ be a node and $k$ be a search key. Then, $succ_{\geq k}(v) = |\{x : x \in v.keys \wedge k > x\}|$, where $|S|$ denotes cardinality of $S$.*

**Algorithm 5.1** Counting search

```
1  int succG (Node *v, int skey) {
2    int count = 0;
3    for(int i=0; i<CAPACITY; i++)
4      count += (skey >= v->keys[i]);
5    return count;
6  }
```

**Algorithm 5.2** SIMD-based counting search (AVX-512)

```
1  int succG_SIMD(Node *v, int skey) {
2    int count = 0;
3    __mmask8 cmp_mask = 0;
4    __512i vec,Vskey = _mm512_set1_epi64(skey);
5    for(int i = 0; i < CAPACITY; i += 8) {
6      vec = _mm512_loadu_epi64((__512i*)(v->keys+i));
7      cmp_mask = _mm512_cmpge_epu64_mask(Vskey, vec);
8      count += _mm_popcnt_u32((uint32_t) cmp_mask);
9    }
10   return count;
11 }
```

and the same code snippets can be used, by replacing comparison `(skey >= v->keys[i])` by `(skey > v->keys[i])` and `_mm512_cmpge_epu64_mask` by `_mm512_cmpgt_epu64_mask`.

The experiment of Figure 5.2 illustrates the efficiency of our data-parallel $succ_{>}$ compared to traditional ways for branching in multiway trees. We created a random array of 64-bit unsigned integers that simulate the array of keys in a full B$^S$-tree node, with values selected uniformly at random from the entire range $[0, 2^{64} - 1]$. Then, we performed 1 million random successor (i.e., branching) operations to the array and measured the throughput (in millions operations per second) of four implementations[1] of the operation:

- **Binary**: use of (non-recursive) binary search
- **Linear**: scan data from the beginning until successor is found
- **Counting**: count-based successor in a for-loop (Snippet 5.1)
- **SIMD-based**: count-based successor using SIMD (Snippet 5.2)

We tested various sizes of the array, modeling different key-array sizes in a B$^S$-tree node. We used array sizes that are multiples of 8, which allows us to take full advantage of SIMD-parallelism. Based on the results, binary search significantly outperforms linear scan in large arrays, but the difference is marginal in small arrays. This is expected as the number of comparisons by linear scan is linear to the array size, whereas binary search incurs a logarithmic number of comparisons. While being faster than linear scan, counting search (Snippet 5.1) also does not benefit from the increase of the array size. Still, it outperforms binary search for array sizes up to 64. This performance is due to the absence of branch instructions, as previously discussed. Additionally, the design of this technique allows the compiler to make optimizations at the assembly level, enhancing the search process through autovectorization and substituting certain instructions by SIMD operations.

Observe the excellent performance of SIMD-based search (Snippet 5.2) for all array sizes, with the exception of 256. Compared to all other methods with black-box compilation, it becomes clear that custom vectorization can offer significant advantages. By tuning the capacity of B$^S$-tree nodes, we can achieve the maximum throughput increase via vectorization compared to traditional implementations. Specifically, for 64-bit keys, the optimal key-array capacity is 16, which achieves 4x performance

[1] See Section 5.6 for our experimental setup.

improvement over binary search, in line with the theoretical expectation ($\log_{16} 16$ vs. $\log_2 16$). As a final note, Snippet 5.2 has double throughput for array sizes 8 to 32 compared to the code produced by compiling Snippet 5.1.

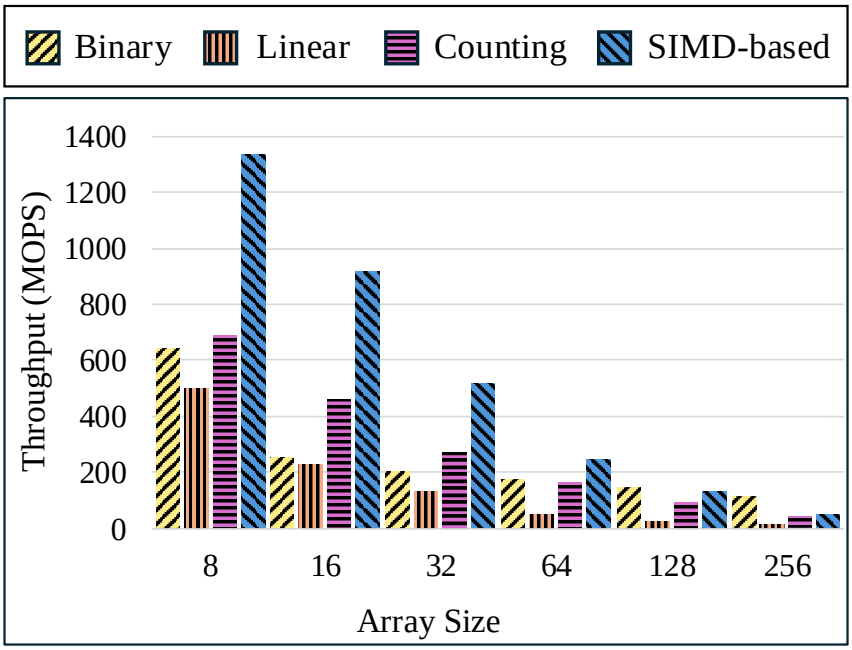



Figure 5.2: Successor search techniques

### 5.1.3 B$^S$-tree search

Algorithms 5.3 and 5.4 show how the B$^S$-tree is searched for (i) equality queries and (ii) range queries. For equality, we traverse the tree by applying a $succ_{>k}(v)$ operation at each non-leaf node $v$. At the reached leaf $v$ we apply a $succ_{\geq k}(v)$ operation to find the first position $r$ in the leaf having a key greater than or equal to $k$. If $v.keys[r]$ equals $k$, then the record at position $r$ is returned; otherwise, $k$ does not exist. Equality search requires one $succ_>$ or $succ_\geq$ operation per node along the search path.

For range queries, assume that we are looking for all keys $x$, such that $k_1 \leq x \leq k_2$. We traverse the tree using $succ_{>k_1}$ operations to find the first leaf that may contain a query result. In that leaf, we apply a $succ_{\geq k_1}$ operation to locate the position $r_1$ of the first qualifying key. To find the end position $r_2$ of the query results, starting from the current leaf $v$, we search for the leaf which includes the first key greater than $k_2$, by performing one $succ_{>k_2}$ operation per leaf. Hence, range searches require one $succ_>$ operation per node along the search path in search for $k_1$ plus one $succ_>$ operation for each leaf that includes range query results. For large query ranges whose results

**Algorithm 5.3** Equality Search

**Input** : search key $k$, B$^S$-tree root node $v$

**Output** : record-id corresponding to key $k$

1: **while** $n$ is non leaf **do**
2: $v \leftarrow$ node pointed by entry at position $v[succ_{>k}(v)]$
3: **end while**
4: $r \leftarrow succ_{\geq k}(v)$ ▷ leaf node
5: **if** $v.keys[r] == k$ **then**
6: **return** record-id in $v$ at positon $r$
7: **else** ▷ $k$ does not exist
8: **return** NULL
9: **end if**

appear in numerous leaves, we apply an alternative implementation of range queries, where one equality search is used to locate $r_1$ and another equality search is then applied to locate $r_2$.

This is expected to be faster than Algorithm 5.4 if the height of the tree is smaller compared to the number of leaves that include the query results.

As an example, consider searching for key 13 in the tree of Figure 5.1. A $succ_{>13}$ operation on the root will give position 0, as there are 0 keys smaller than or equal to 13, so the first pointer of the root will be followed. Then, the $succ_{>13}$ operation on the visited node will return 1, which means that we then visit the 2nd leaf, where $succ_{\geq 13}$ is applied that returns 1, i.e., the position of 13 in the leaf. A range search for keys in $[13, 17]$ first locates 13 and then finds the upper bound 18 in the same leaf after applying $succ_{>17}$. Another example about range query, find records with keys in $[13, 22)$, we apply the same search (i.e., find the smallest key larger than 13 at non-leaf nodes, find the smallest key greater than or equal to 13 at the leaf node) to locate the first query result. From thereon, we scan the node and its siblings (using the chain pointers) and output results as long as they are within the query range. To minimize comparisons, at each leaf we can search for the first key which is greater than the end of the query range.

**Algorithm 5.4** Range Search

**Input** : search keys $k_1, k_2$, B$^S$-tree root node $v$

**Output** : record-ids of keys $x$, where $k_1 \leq x \leq k_2$

1: **while** $n$ is non leaf **do**
2: $v \leftarrow$ node pointed by entry at position $v[succ_{>k_1}(v)]$
3: **end while**
4: $r_1 \leftarrow succ_{\geq k_1}(v)]$
5: **while** $(r_2 \leftarrow v.keys[succ_{>k_2}(v)]) ==$ N **do**
6: $v \leftarrow nextleaf(v)$ ▷ next to $v$ leaf
7: **if** $v ==$ NULL **then**
8: **break** ▷ last leaf reached
9: **end if**
10: **end while**
11: **return** record-ids of keys from position $r_1$ to position $r_2$ (excl.)

## 5.2 Updates

Another important novelty of B$^S$-tree is the handling of updates. As discussed, unused key slots at the end of each node are filled with MAXKEY values; hence, uniqueness is not a requirement for unused key positions. In addition, B$^S$-tree does not require the used key slots to be continuous from the beginning of the node. This means that 'gaps' with unused key slots are allowed in a node. The key in a gap is the same as the *first subsequent non-gap key*. By managing auxiliary information at each node (i.e., number of used slots, bitmap indicating used slots), we can efficiently track unused slots regardless of the data distribution within the node (see Figure 5.3 for an example). In the rest of this section, we will show how deletions and insertions are handled in the B$^S$-tree. We will explain how our approaches minimize the overhead of node modifications while maintaining high search performance.

### 5.2.1 Deletions

To delete a key, we first locate its position $i$ in a leaf node, using the equality search algorithm (discussed in Section 5.1.3). To conduct the deletion, we copy the key value from position $i+1$ to position $i$ and propagate it backwards to previous gap positions in the node. If $i$ is the last position in the leaf, we set $v \rightarrow keys[i]$=MAXKEY. One

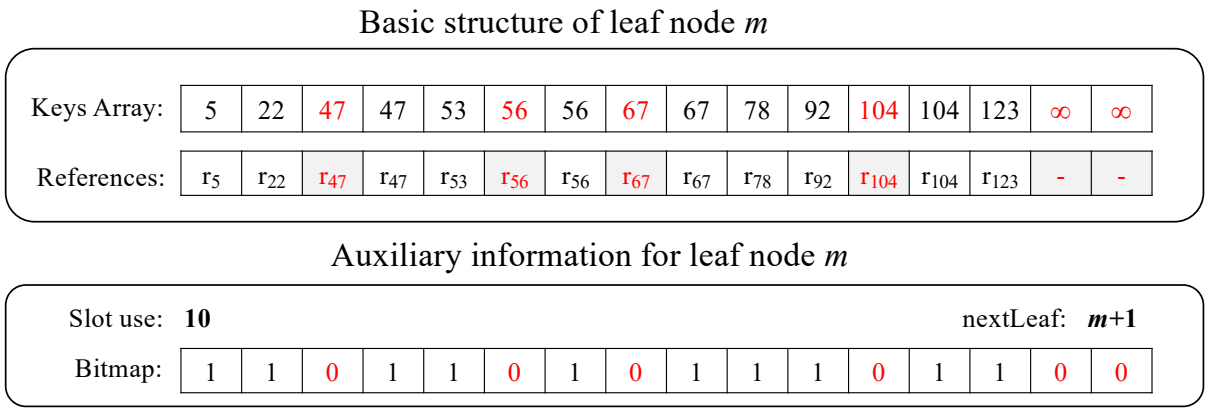



Figure 5.3: B$^S$-tree leaf node structure

subtle point to note is that $succ_{\geq}$ may not give us the real position of the key $k$ to be deleted but may give us the first of a sequence of gaps that have key value equal to $k$. For example, for deleting $k = 56$ in Figure 5.3, we apply $succ_{\geq}(56)$ which gives us position 5. Then, we find the range of all positions having 56 (i.e., [5,6]) and copy into them the next value (i.e., 67). Finding all the positions can be done very fast using bitwise operations. Figure 5.4 (top) shows the leaf node of Figure 5.3 after deleting key 56. Algorithm 5.5 is a pseudocode for deletions to the B$^S$-tree.

As in previous work [64, 39], we do not handle underflows and allow nodes with fewer than 50% occupied slots. The reason is that insertions are anticipated to be more frequent than deletions in workloads, so node merges or key redistributions are not expected to pay-off in the long run. If a node becomes empty after a deletion, the node is removed and the corresponding separator entry at its parent is also 'deleted' by copying the next key into it.

### 5.2.2 Insertions

Inserting a new key $k$ to the B$^S$-tree entails searching for the leaf node and the position in it to place it. Search is conducted by applying $succ_{>k}$ operations starting from the root and following the corresponding pointers. When we reach the leaf $v$ where in $k$ should be inserted, we apply a $succ_{\geq k}$ operation which finds the proper slot in $v$ to insert $k$. Then, we verify whether the slot $i$ returned by $succ_{\geq k}$ is occupied by another key. This is done by a simple test. If $v \rightarrow keys[i] = v \rightarrow keys[i+1]$, then we are sure that position $i$ is free, so we can place $k$ there and finish. For example, assume that we want to insert key 55 to the leaf node of Figure 5.3. We apply a

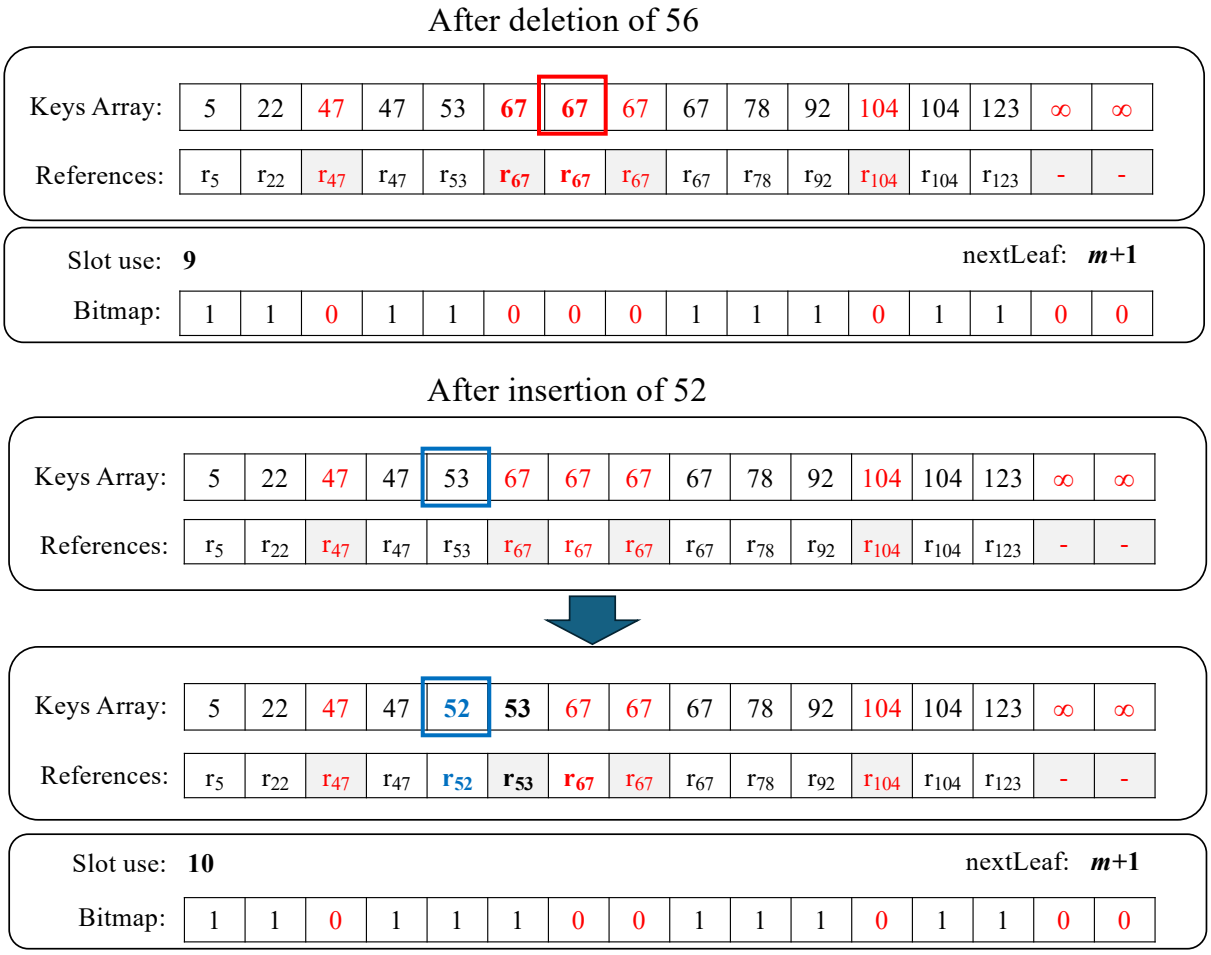



Figure 5.4: Updates to B$^S$-tree leaf node

$succ_{\geq k}$ operation to the leaf, which will give us position 5. Since the next position (6) has the same key, position 5 corresponds to a free slot (gap), hence, we have directly put the inserted key 55 there. On the other hand, if the key at position $i$ is different compared to the key at position $i+1$, this means that the position is occupied. In this case, we first find the first position $j$ after $i$, which is unused (i.e., a gap), and right-shift all keys (and the corresponding record pointers) from position $i$ to position $j-1$, to make space, so that key $k$ can be inserted at position $i$. If there is no free position after $i$, then we move one position to the left (left-shift) all keys and record ids from position $i$ until the first free position to the left of $i$. Figure 5.4 (bottom) shows an example of inserting key 52. As the slot where 52 should go is occupied by 53 and it is not a gap (the key following 53 is not equal to 53), we search for the next gap, which is the position next to 53, right-shift 53 there and make space for the new key 52. Algorithm 5.6 describes the insertion procedure to a B$^S$-tree.

In case leaf $v$ is full, then we conduct a *split* of $v$ and introduce a new leaf node.

The existing keys in $v$ together with $k$ are split in half and distributed between the two leaves. Instead of placing the distributed keys to the first half of each of the two leaves, we interleave each key with a gap to facilitate fast insertion of future keys. Proactive gapping is described in the next subsection.

### 5.2.3 Construction

We now describe the algorithm for building a B$^S$-tree from a set of keys (bulk loading). Like typical B$^+$-tree construction algorithms, we first sort the keys to construct the leaf level of the index. To facilitate fast future insertions, we do not pack the nodes with keys, that is we leave free space to accomodate future insertions. Specifically, if $N$ is the capacity of a leaf node, we construct all leaf nodes by adding to them the keys in sorted order; each leaf node takes $\alpha \cdot N$ (key, record-id) pairs, where $\alpha$ ranges from $0.5$ (half-full nodes) to $1$ (full nodes). We typically set $\alpha = 0.75$. For each leaf, instead of placing all keys at the beginning of the leaf and leaving $(1-\alpha)N$ consecutive empty slots at the end of the node, we *spread* the entries in the leaf by placing one gap (empty slot) after every $\frac{1}{1-a} - 1$ entries.[2] For each leaf node (except the first one) a *separator key*, equal to the first key of the leaf, is added to an array. For each separator key, a node pointer to the previous leaf is associated to the separator. Finally, a node pointer to the last leaf is introduced at the end of the array (without a key value). After constructing the leaves, the (already sorted) array of separator keys is used to construct the next level of B$^S$-tree (above the leaves), recursively.

## 5.3 Key Compression

In real-world database applications that rely on in-memory indexing, minimizing resource utilization is crucial for supporting large-scale addresses. Taking into consideration this challenge, we propose a variant of B$^S$-tree that focuses on reducing the memory footprint while aiming to have as little impact as possible on search efficiency. Previous work [40] has introduced a novel approach to optimizing B-trees in main-memory databases through key compression using fixed-size partial keys. This method specifically aims to mitigate cache misses, which are a common bottleneck in tree traversal operations, while simultaneously achieving a significant reduction in

[2]Gaps between consecutive key values (for integer keys) are not introduced.

memory consumption. However, reliance on partial keys necessitates a decompression mechanism when their bit representations lack uniqueness, potentially introducing overhead during search operations. To avoid the trade-off between memory efficiency and performance, we propose a novel method - to our knowledge - that eliminates issues related to reconstruction. For B$^S$-tree, we opt for a simpler key compression technique, which incurs minimal search overhead.

For each node $v$, we store in the node's auxiliary information (see Figure 5.3) the first key $v.k_0$ of the node and replace the $v$'s key array (of size $N$) by an array where each original key $k$ is replaced by the difference $k - k_0$. This allows us to potentially double or quadruple the size of the array if the differences occupy much less space than the original keys. If $N = 16$ and the original array stores 64 bits, it may potentially be replaced by an array of $N = 32$ 32-bit differences or $N = 64$ 16-bit differences. Since the keys in a node are ordered, we expect the differences to be small, especially in leaf nodes, so the space savings due to the reduction in the number of nodes are expected to be significant. To achieve optimal performance of our data-parallel $succ_>$ implementation, we set the key array size to 1024 bits, so $N$ can be 16, 32, or 64. Another benefit of compression is that a potentially radical decrease in the number of (leaf) nodes may reduce the number of tree levels, rendering the tree faster.

The variability in the capacity of nodes necessitates some modifications to the auxiliary information at each node. Previously, 8 bytes were sufficient to store all relevant details about slot use, the bitmap of a node, and the next leaf reference. While the number of bits for slot use and the next leaf reference can still be represented as before, this is not the case for the bitmap. Since the bitmap corresponds to the total number of entries in a node, we have opted to allocate 64 bits for this purpose, which is sufficient for any type of leaf. We now discuss the impact of this scheme in the tree operations.

**Tree construction** Our goal is to construct the tree in one pass over the sorted keys and to result in leaf nodes having 75% occupancy (except when we are dealing with regions of sequential key values), while achieving the best possible compression. For this, we begin by checking whether the leaf can be filled with 16-bit differences for the keys. If this is not feasible, we reattempt the process by checking if half of the keys can be stored as 32-bit differences. If this attempt also fails, we conclude by storing the exact 64-bit keys.

**Search** To apply $succ_{>k}$ at a node $v$ we first compute $k' = k - v.k_0$, where $v.k_0$ is the first key value of $v$, stored explicitly in $v$'s meta-data. Then, we apply $succ_{>k'}$ to the node to find the position of the node pointer to follow. The same procedure is applied at the leaf nodes for $succ_{\geq k}$; the position of $k' = k - v.k_0$ corresponds to the position of $k$, or if $succ_{\geq k'}$ returns NULL, $k$ does not exist.

**Insert** We can directly use the B$^S$-tree insertion algorithm to insert a new key $k$, by first running the search algorithm discussed above to find the leaf $v$ and the position in $v$ whereto insert $k$ and then store the difference $k - v.k_0$ there. Keys can be stored exactly only as first keys in new nodes that are constructed after a split. The new nodes after a node $v$ is split can be of the same type as $v$, or they can be further compressed as they include fewer entries than $v$ with the first and the last one having smaller differences.

**Delete** Deletion is not affected by key compression. When the key to be deleted is found (represented exactly or by its difference to $k_0$) at the corresponding leaf node, we simply copy into it the value of the next key, or MAXKEY if the deleted key is the last one in the leaf node. In compressed nodes, MAXKEY is the maximum value that can be represented using all available bits. If the first key of a node is deleted, we do not change $k_0$ (as it is not stored in a slot of the array) and keep in the slots the differences to $k_0$.

## 5.4 Implementation Details

This section presents some implementation details of the B$^S$-tree that have a significant impact in its performance in practice.

**Node size and structure.** As in previous work [67, 43, 45, 46, 54, 47, 53, 55, 56, 60, 64, 66], we aim at indexing large keys, each being a 64-bit unsigned integer. To take full advantage of our SIMD $succ_>$ implementation, each node stores a maximum of 16 entries (see the experiment of Figure 5.2); based on this, we allocate $16 \times 64 = 1024$ bits for the keys of each node. Hence, the keys of each node (internal or leaf) fill two cache lines (each cache line can store 64 bytes). This means that for $succ_>$, we perform 2 SIMD instructions per node by loading the keys at 2 registers of 512 bits (8 keys at each register). $1024$ bits are also allocated for keys in the compressed CB$^S$-tree nodes, so a compressed key array may have 32 32-bit entries (key differences) or 64

16-bit entries. For gap management, we use a 32-bit variable which keeps the slot use and bitmap of each node. For the leaves, we have an additional 32-bit field for the next leaf address. As discussed in Section 5.2.3, gaps are added to the tree nodes proactively at construction time, to allow efficient ingestion of insertions. Since we anticipate insertions to affect mainly the leaves, in practice, we use a much smaller percentage of gaps at inner nodes (one), to keep the height of the tree small.

**Memory management.** In this section, we will describe the structure of our $B^S$-tree. The $B^S$-tree is an implicit structure that uses only keys (no key-value store) with the goal of improving memory consumption, operation performance, and cache efficiency. As we have already determined, the optimal block size for our index is 16. To store the $B^S$-tree in memory, we utilize two main structures: one to store the inner nodes and another for the leaf nodes. Each inner node consists of two arrays with 16 entries. The first array holds 64-bit keys, while the second contains 32-bit references to nodes. 32 bits are sufficient for the references because they are in fact offsets to fixed-length slots in memory arrays allocated for nodes (one for inner nodes and one for leaf nodes). The auxiliary data for each node are put in a separate dedicated array aligned with the node arrays. Hence, each inner node has a size of 192 bytes, which fits into 3 cache lines. Our tested $B^S$-tree implementation only has keys and not values (i.e., record-ids) in its leaves, so a leaf node contains a single array of 16 64-bit keys, with each leaf node occupying 128 bytes, fitting into 2 cache lines. The inner nodes are stored in a contiguous array, aligned to Transparent Huge Pages (2 MB) for efficiency. The leaf nodes are also stored in a contiguous array, but their alignment depends on the array size. If the array is smaller than 3 GB, we align it using huge pages. Otherwise, it is aligned per cache line (64 bytes). Alignment plays a crucial role in optimizing both cache efficiency and the use of SIMD operations, making it a key factor in the performance of $B^S$-tree. By aligning data to cache lines, we minimize cache misses and ensure that the CPU can retrieve entire nodes in a single memory access, significantly speeding up operations. By aligning inner nodes to huge pages (2 MB), we reduce translation lookaside buffer (TLB) misses. For larger datasets, aligning leaf nodes per cache line helps manage memory effectively without sacrificing access speed. We also make use of __builtin_prefetch, a compiler intrinsic that allows us to pre-load data into the cache before it is needed, reducing latency. By prefetching data in anticipation of future operations, we ensure that memory access is even more efficient, further boosting the performance of key operations such as

searches and inserts. By combining cache-line and SIMD-friendly alignment, along with appropriate use of __builtin_prefetch, B$^S$-tree allows for SIMD acceleration, and reduces memory access latency, leading to significantly better overall performance.

**Compress or not?** The compressed version of B$^S$-tree with variable-capacity nodes (Section 5.3) may reduce the memory footprint of the index and improve its performance, but also comes with the overhead of explicitly keeping the first key of a node, which does not pay off for nodes having 64-bit differences that cannot be compressed.[3] Hence, we employ a *decision mechanism* for choosing between the construction of a B$^S$-tree or a compressed B$^S$-tree, based on the input data. The key idea is to quickly and easily assess whether a dataset can be stored using compressed information (i.e., keys smaller than 64 bits). This mechanism operates during the construction phase, before the leaf nodes are created. Before bulk-loading the tree, we virtually split the sorted keys input into segments of 13 keys each, subtract the smallest key from the largest key in each bucket, and calculate the number of leading zeros. After performing these calculations for all segments, we take the average number of leading zeroes. If this average is greater or equal to 32 bits, we conclude that the dataset can benefit from a compact B$^S$-tree compression, and we go ahead with its construction. Otherwise, we create a standard (uncompressed) B$^S$-tree. The selection of 13 keys is not arbitrary, as we put 25% gaps at each leaf, and the 13th key serves as the separator for the node. By using this mechanism, we avoid manually choosing which version of the B$^S$-tree to implement, instead allowing our algorithm to automatically select the appropriate We found out that compression is not effective for inner nodes, so our final compressed B$^+$-tree implementation has uncompressed inner nodes and compressed leaves.

## 5.5 Concurrency control

A wide array of concurrency control techniques has been proposed for the B$^+$-tree and similar indices. Lock coupling [168, 117] is among the earliest techniques enabling concurrent operations in a multi-threaded environment. The core principle of this technique is to hold a lock on the current node only long enough to ensure a safe transition to the next node, at which point the lock is acquired for the next node and

[3] As we want all leaf nodes to have the same fixed size (for alignment purposes), we do not allow the same B$^S$-tree to have both uncompressed and compressed leaves.

released for the previous one. However, a key drawback is that locks must be continually acquired and released at each step, which can impair performance. The B-link tree [169] introduces right-sibling pointers at each node, connecting to neighboring nodes on the same level. This pointer mechanism ensures that traversals proceed accurately even when the tree is being modified, though it introduces increased memory overhead and requires additional operations during node splits. Bronson et al. [170] proposed a concurrency technique for binary search trees that combines fine-grained locking with lock coupling and logical deletions. This approach allows for efficient concurrent operations, including search, insertion, and deletion, while minimizing contention. The Bw-tree's concurrency control mechanism [43, 171] is lock-free, employing a mapping table that provides an indirection layer for atomic, non-blocking updates. Delta records allow for incremental modifications without locking. Despite these advantages, this technique's drawbacks include delta chain overhead, complexity, and increased memory usage. Read-Optimized Write Exclusion (ROWEX) [149] is another concurrency protocol, employed in ART [53] and HOT [55]. It supports non-blocking reads, requiring locks only for writes. Writers must ensure consistency by using atomic operations. While promising, this approach can be challenging to implement across various data structures and often demands significant code adjustments. Leis et al. [172, 149] introduced the Optimistic Lock Coupling (OLC) technique for B-trees and $B^+$-trees, which is straightforward to implement, highly efficient, and delivers strong performance. In OLC, when a thread seeks to read or modify a node, it first acquires an optimistic read lock, allowing it to traverse the tree while maintaining a local copy of the node's state. For updates, the thread verifies whether the node has been modified by another thread since it was read. If not, it applies the update and commits atomically. If a conflict arises (i.e., another thread modified the node), the thread discards its changes and restarts the operation from the root. Our current implementation of $B^S$-tree does not include a concurrency control mechanism. Since our $B^S$-tree is essentially a $B^+$-tree, OLC is the most suitable and efficient concurrency technique for it and we plan to implement it in the future.

## 5.6 Experiments

In this section, we experimentally compare $B^S$-tree and its compressed version to alternative main-memory indices (learned and non-learned). As in previous work [55,

53], we have built and compared indices for key data only; record ids or references are not stored in each index, but the objective of each method is to locate the position(s) of the searched key(s). The implementation of all methods is in C++ and compiled with `gcc` (v13) using the flags `-O3` and `-march=native`. The experiments were conducted on a system with an 11th Gen Intel® Core™ i7-11700K processor running at 3.60 GHz, 128 GB of RAM, and AVX 512 support. The operating system used was Ubuntu 20.04.

### 5.6.1 Setup

**Datasets**. We ran our tests on standard benchmarking real datasets used in previous work [156, 157, 158]; each on consists of unsigned 64-bit integer keys (potentially reduced to 63-bit, as HOT cannot handle 64-bit keys). In Amazon BOOKS [156, 157], each key represents the popularity of a specific book. In FB [156, 157, 173], each key is a Facebook user-id. OSM [156, 157]

contains unique integer-encoded locations from OpenStreetMap. GENOME [158, 174] includes loci pairs from human chromosomes. PLANET [158, 175], a planet-wide collection of integer-encoded geographic locations compiled by OpenStreetMap. We preprocessed the datasets to eliminate any duplicates. According to [158, 66], OSM, FB, GENOME, and PLANET are complex real-world datasets that can pose challenges for learned indices. In contrast, the key distribution of BOOKS is easy to learn. We did not conduct experiments using synthetic datasets with common distributions, as, according to [156], it would be trivial for a learned index to model such distributions.

**Competitors**. We compare our proposed **B$^S$-tree** and its compressed version, denoted by **CB$^S$-tree**, with five updatable learned and non-learned indices, for which the code was publicly available by the authors (we thank them!). The selection was done based on the superiority of these indices compared to alternative methods (e.g., ART [53] and other updatable learned indices).

**Non-learned Indices**. STX library [176] is a fully optimized C++ implementation of a main-memory B$^+$-tree. We use the set-based implementation from STX, which does not store values in the leaf nodes. For its construction, we used its fast bulk-loading method. We used the default block size of STX (256 bytes), so each leaf node holds 32 keys ($32 \times 8 = 256$ bytes), and each inner node holds 16 keys and 16 pointers ($16 \times 8 + 16 \times 8 = 256$ bytes). We used two versions of the STX tree: the first is the

original code, referred to as **B$^+$-tree**, while the second version creates 25% empty space at the end of each leaf node, denoted by **Sparse B$^+$-tree** (for fairness, as our B$^S$-tree also proactively introduces gaps at its construction to support fast insertions). STX supports $2^{64}-1$ values in keys. The implementation of **HOT** [177, 55] stores only keys and does not support bulk-loading. However, we observed that by pre-sorting the data, HOT's construction time improves and we also get a more efficient trie. The HOT code release does not include a built-in range query implementation, so we implemented one, following B$^S$-tree's logic. HOT cannot handle keys greater than $2^{63}-1$, so we removed values exceeding this limit from certain datasets.

**Learned Indices.** **ALEX** [178, 60], **LIPP** [179, 64], and **DILI** [180, 65] are all learned indices for key-values. To use them, for each dataset we used as value of each key the key itself, during construction and insertions (writes). They all support bulk-loading. ALEX and LIPP can handle 64-bit keys, i.e., (unsigned) integers up to $2^{64}-1$. DILI could not run on two of our five datasets.

**Workloads.** We used several different workloads used to measure throughput. First, we randomly selected 150 million entries from each dataset for the construction phase. where we sorted the data and applied bulk-loading (except for HOT, which, however, benefits from sorting). For our workloads, we used 50 million keys, that are selected randomly (i.e., queries and updates hit a random region of the space). Our workloads are:

- **Read-Only (Workload A)**: 100% reads (equality searches).
- **Write-Only (Workload B)**: 100% writes (insertions).
- **Read-Write (Workload C)**: 50% reads, 50% writes.
- **Range-Write (Workload D)**: 95% range searches, 5% writes.
- **Mixed (Workload E)**: 60% reads, 35% writes, 5% deletions.

### 5.6.2 Construction Time and Memory Footprint

In this section, we compare all tested methods with respect to their construction cost and memory footprint. From each dataset, we used 150 million keys to construct each index. For the evaluation of the experiments we use datasets consisting of 150 million keys from Amazon Books, OSM, Facebook, Genome, and Planet datasets.

Most of our competitors support bulk loading (ALEX, LIPP, DILI, B$^+$-tree), with HOT being the only exception. However, we observed that if we first sort the dataset before insertion, HOT achieves better construction times and creates a more efficient index. Since all indices require the data to be sorted (or benefit from sorting), we exclude sorting from the construction cost. Table 1 presents the construction times, while Table 2 shows the memory footprints. The construction time of our B$^S$-tree also includes the decision-making mechanism (roughly takes 0.03 sec) on whether we will construct a B$^S$-tree or a CB$^S$-tree (see Section 5.4). To calculate the memory usage of each method, we utilize the C function `getrusage`[4]. Since all learned indices essentially store 64-bit values together with the keys, we report half of their measured memory requirements, to approximate the memory required just for the keys and the models (and inner structure) that they use.

As expected, non-learned indices (except from the CB$^S$-tree) have a stable construction time and memory footprint, which is expected since their construction and memory does not depend on data distribution. On the other hand, the construction time of learned indices can vary significantly, as different data distributions can greatly affect them. As expected, the construction cost of the Sparse B$^+$-tree is higher compared to that of the B$^+$-tree and the two versions of B$^+$-tree there is also a similar-scale difference in their memory footprints. As we will show later, Sparse B$^+$-tree has a performance advantage over B$^+$-tree for any workload that includes write operations. Our B$^S$-tree (and its compressed CB$^S$-tree version) is faster to build compared to B$^+$-tree and Sparse B$^+$-tree mainly due to our better memory management (static pre-allocation vs. dynamic allocation) and because we use offset addressing instead of memory pointers (see Section 5.4).

CB$^S$-tree has the smallest memory footprint than all methods for FB, GENOME, and PLANET because of its high compression effectiveness, which also has a positive impact to the construction time. On the other hand, CB$^S$-tree occupies more space than B$^S$-tree on BOOKS because the distribution of keys there does not provide many compression opportunities; in this case, most leaf nodes store 64-bit differences and also need to explicitly store the 64-bit key of the first key, which renders the size of the index even larger than that of B$^S$-tree. Note that for BOOKS and OSM, our decision mechanism (see Section 5.4) chooses to construct a B$^S$-tree while for FB, GENOME, and PLANET it decides to construct a CB$^S$-tree.

[4]https://man7.org/linux/man-pages/man2/getrusage.2.html

HOT, requires significantly more time to build compared to our methods and B$^+$-tree. The main issue with HOT is that it is a top-down trie and lacks a bulk-loading mechanism, requiring keys to be inserted one at a time. HOT uses slightly less memory than B$^S$-tree; however, our CB$^S$-tree has a significantly smaller memory footprint than HOT in three out of the five datasets.

Regarding learned indices, DILI's offline construction time is significant, as the algorithm first needs to build a BU-tree on disk and then read it to construct DILI. ALEX and LIPP also have high construction time compared to the versions of B$^+$-tree and B$^S$-tree, due to the overhead of training models that predict key positions. Note that all versions of B$^+$-tree and B$^S$-tree have smaller memory footprints compared to the learned indices. Among learned indices, LIPP is the fastest one to construct, because it does not readjust the models when conflicts occur (two or more keys are mapped to the same position). To address conflicts, LIPP creates new nodes, speeding up bulk-loading. ALEX has higher construction time because it frequently adjusts its learned models to preserve key prediction accuracy. This results in computationally expensive node splits and reorganizations. LIPP's downside is its large memory footprint, caused by creating new nodes during conflicts. Node reorganization in ALEX requires less memory. The significant memory overhead of LIPP has also been coined in other experimental studies [158, 66].

In conclusion, the B$^S$-tree has low construction cost, with the B$^+$-tree exhibiting comparable performance alongside a very small memory footprint. Additionally, the CB$^S$-tree achieves the fastest construction time and consumes from 56% to 94% less memory than all methods in FB, GENOME, and PLANET. Our results align with the findings [158], indicating that memory efficiency is not a distinct advantage of updatable learned indices.

### 5.6.3 Throughput

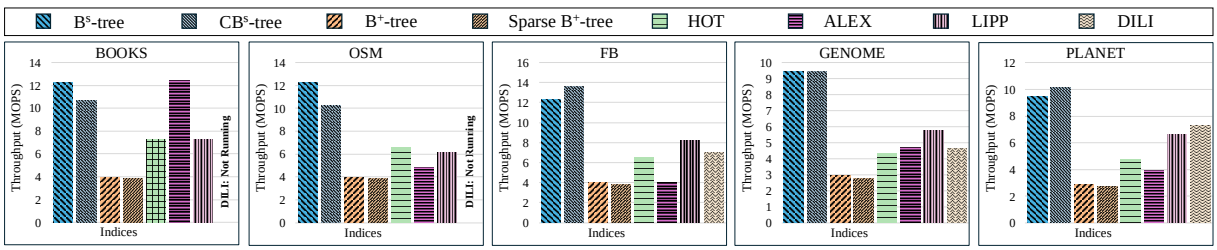



Figure 5.5: Workload A : Read Only (100%)

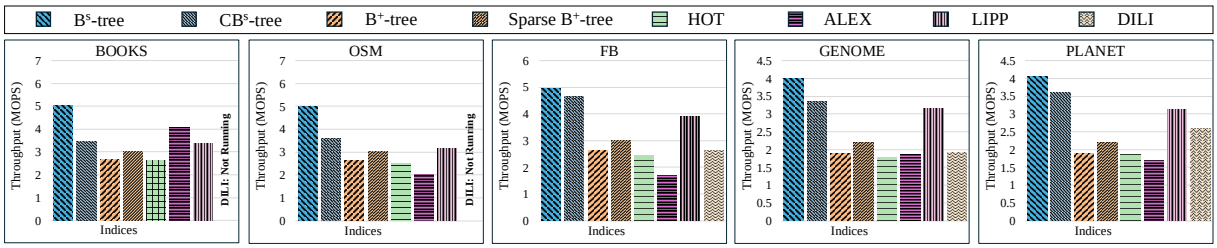



Figure 5.6: Workload B : Write Only (100%)

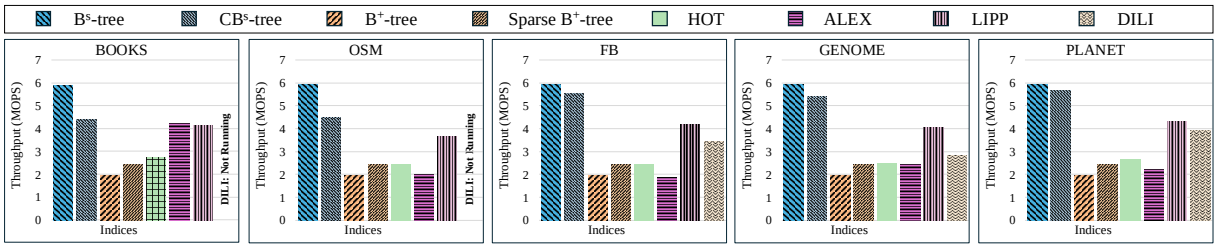



Figure 5.7: Workload C : Read (50%) - Write (50%)

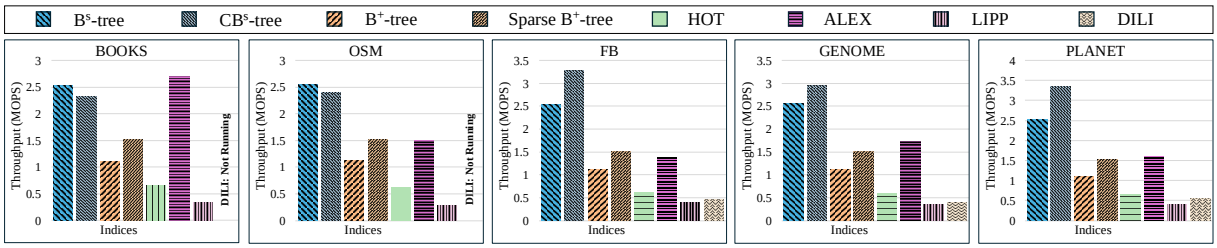



Figure 5.8: Workload D : Range (95%) - Write (5%)

Next, we evaluate the throughput of all methods on the five workloads described in Section 5.6.1. As mentioned, DILI cannot run on BOOKS and OSM datasets, so we include an empty bar labeled "DILI: Not Running" to denote its unknown throughput.

Figure 5.5 presents the throughput (millions of operations per second) of all methods for Workload A (read-only). B$^S$-tree and CB$^S$-tree outperform all competitors across the board except for BOOKS, where ALEX is marginally faster than B$^S$-tree. The excellent performance of ALEX on BOOKS is due to its smooth distribution which is easy for ALEX to learn. In general, learned indices are known to perform well when applied on easy-to-learn CDFs. On average, our methods have a significant performance gap compared to the nearest competitor. Specifically, B$^S$-tree is roughly 2x faster than HOT on OSM, 1.5x faster than LIPP on FB and GENOME, and 1.5x faster than DILI on PLANET. CB$^S$-tree is about 14% slower than ALEX on BOOKS, but much faster than previous work on all other datasets and even faster

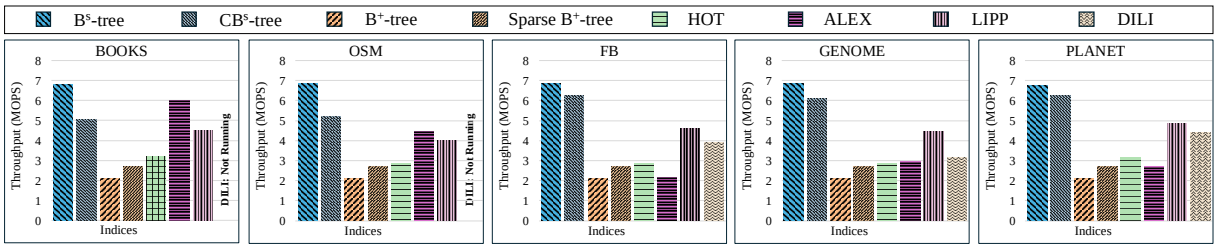



Figure 5.9: Workload E : Read (60%) - Write (35%) - Deletions (5%)

than B$^S$-tree on FB, GENOME, and PLANET, while having a much smaller memory footprint. CB$^S$-tree exploits the highly compressible keys of FB, GENOME, and PLANET to drastically reduce the capacity of leaf nodes and the overall space required for the index. This increases the likelihood that multiple searches hit the same leaves, exploiting the memory cache, as we will also show in the next set of experiments. LIPP and DILI outperform ALEX on datasets other than BOOKS because their models are precise (i.e., ALEX applies exponential search at the last mile, when its prediction is imprecise).

For Workload B (write-only), as Figure 5.6 shows, B$^S$-tree outperforms all methods, while CB$^S$-tree loses to ALEX only on BOOKS. Compared to LIPP and DILI, ALEX performs worse on the other four datasets because It frequently needs to readjust its learned models to sustain key prediction accuracy, resulting in computationally expensive node splits and reorganizations. LIPP is more robust than other learned indices due to its accurate prediction mechanism, which reduces the need for frequent node splits or rebalancing due to the more appropriate placement of keys during its construction. Observe that the Sparse B$^+$-tree is faster than the B$^+$-tree because it requires much fewer splits. CB$^S$-tree is slower than B$^S$-tree, because splits are more costly for compressed nodes.

The results on Workload C (read-write) are presented in Figure 5.7. In this workload, our algorithms outperformed all competitors across all datasets. In general, the performance of all methods stands between that of Workload A (read-only) and Workload B (write-only), which is expected.

For the results of Workload D (range-write), see Figure 5.8. The results are similar compared to Workload A (read-only) since Workload D is read-heavy. Range queries retrieve 153 keys on average. ALEX has a better performance compared to other learned indices for range queries, as the structures of LIPP and DILI are not optimized for range scans. ALEX can have large nodes with more sequential keys, allowing

it to avoid jumping to sibling nodes. HOT is also not optimized for range scans, therefore its low compared to other workloads. On the compressible datasets (FB, GENOME, PLANET), CB$^S$-tree outperforms B$^S$-tree due to the smallest number of memory accesses it requires to obtain the results of range queries, as it reads numerous consecutive compressed leaf nodes.

Figure 5.9 shows the results using Workload E (read-write-delete). The results are similar to those for Workloads A and B. B$^S$-tree and CB$^S$-tree outperform all competitors across all datasets except for BOOKS, where CB$^S$-tree is slightly inferior to ALEX. Deletions do not impose an overhead to B$^S$-tree and all other methods.

### 5.6.4 Performance Counters

Besides throughput, we also compared all methods with respect to various performance counters. The metrics used for comparison include instructions executed, cycles, mispredicted branches, and misses in L1, LLC (Last Level Cache), and TLB (Translation Lookaside Buffer) misses. As representative datasets, we selected BOOKS and FB; ALEX exhibits best on BOOKS, while on FB our methods (B$^S$-tree and CB$^S$-tree) have the best throughput compared to the competitors. We chose to measure Workload C, as it is update-heavy and the unpredictable nature of writes can lead to numerous splits, stressing the indices. To calculate these metrics, we utilized Leis's perf_event code [181]. As mentioned, DILI cannot run on BOOKS. The average performance measures per operation for all methods are presented in Tables 5.3 and Table 5.4 for BOOKS and FB, respectively.

B$^S$-tree needs the smallest number of instructions and cycles, with CB$^S$-tree being closed to the runner up. This demonstrates that our algorithms are simple and efficient, benefiting from the use of SIMD instructions and alignment, which reduce the number of cycles and instructions required for each task. In terms of mispredicted branches, our algorithms also have the lowest values for both datasets, which is expected since our search code is branchless. All mispredicted branches in B$^S$-tree arise from the insertions. Regarding L1 and LLC, on average, our algorithms incur 16 cache misses, which is consistent with our expectation. Specifically, the height of our trees is 6 and their fanout is 16; we encounter 2 cache misses per tree level and 2 cache misses at the leaf level ($2 \times 6 + 2 = 14$ misses), with the remaining 2 misses caused by insertions. For BOOKS, LIPP achieves the best performance in terms of

L1 and LLC cache misses, though our algorithms are close in comparison. For the FB, DILI performs the best for L1 misses, while CB$^S$-tree achieves the best overall performance. Overall, we expected to have slightly higher L1 and LLC misses due to our tree's greater height compared to learned indices. Lastly, in terms of TLB misses, our algorithms exhibit outstanding performance relative to our competitors. This can be attributed to our use of huge pages (see Sec. 5.4). Overall, the performance counters show that our B$^S$-tree and CB$^S$-tree are fully optimized via the use of SIMD instructions, huge pages, and branchless code, aimed at creating an updatable, efficient, and cache-friendly index.

### 5.6.5 Summary of Experimental Findings

In summary, B$^S$-tree and CB$^S$-tree exhibit excellent and robust performance for different workloads and different datasets of varying distribution, being superior than all competitors in most cases. Note that our decision mechanism, which imposes a small overhead in the construction (up to 10% of the construction cost) decides automatically which of B$^S$-tree or CB$^S$-tree to build for a given dataset. The decision is B$^S$-tree for BOOKS and OSM and CB$^S$-tree for FB, GENOME, and PLANET. Although CB$^S$-tree is not superior to B$^S$-tree on all workloads over FB, GENOME, and PLANET, their difference of the two is not high on these datasets, while CB$^S$-tree is much faster to build and has a much lower memory footprint.

Regarding updatable learned indices, our study shows that they can be outperformed by data-parallel non-learned indices optimized for main-memory, such as our B$^S$-tree. From the tested methods, LIPP appears to be superior and more robust than ALEX and DILI, except for range workloads, where ALEX performs best. In addition, their construction cost is high and they have large memory footprint (except for ALEX) compared to non-learned indices. Even on the BOOKS, which has a easy-to-learn key distribution, our B$^S$-tree demonstrates competitive performance in searches and range scans compared to ALEX and outperforms it in all other workloads.

## 5.7 Conclusions

We proposed B$^S$-tree, an efficient main-memory implementation of the B$^+$-tree that uses a simple and intuitive data-parallel implementation for branching at each level

during search and update operations. We propose a novel way for implementing gaps (unused slots) at nodes by copying into them the next used key, which does not affect the functionality and efficiency of our branching operator. Finally, we propose a compression mechanism for B$^S$-tree nodes that allows them to have variable capacity based on the allowed room for compression. Our experimental evaluation demonstrates the superiority of B$^S$-tree compared to open-source state-of-the-art non-learned and learned indices, with respect to construction time, memory footprint, and throughput for various workloads that include queries and updates.

**Algorithm 5.5** Deletion in $B^S$-tree

**Require**: key $k$, $B^S$-tree root node $v$

1: find leaf $v$ and position $r$ by running lines 1-3 of Alg 5.3
2: **if** $v.keys[r] \neq k$ **then**
3: **return FAIL**
4: **end if**
5: $bitmap \leftarrow v.bitmap$
6: **if** $r == N - 1$ **then** ▷ last key in node
7: $bitmap \leftarrow bitmap \oplus$ `0x0001`
8: $replicasOfKey \leftarrow \_tzcnt(bitmap)$
9: **for** $i \leftarrow 0$ **to** $replicasOfKey - 1$ **do** ▷ propagate backwards
10: $v.keys[r - i] \leftarrow MAXKEY$
11: **end for**
12: **else** ▷ $r$ is not the last position in the leaf
13: $bitmap \leftarrow bitmap \oplus$ (`0x8000` $\ll r$)
14: $replicasOfKey \leftarrow \_lzcnt(bitmap)$
15: $nextValidKey \leftarrow v.keys[r + replicasOfKey + 1]$
16: **for** $i \leftarrow 0$ **to** $replicasOfKey$ **do** ▷ propagate backwards
17: $v.keys[r + i] \leftarrow nextValidKey$
18: **end for**
19: $bitmap \leftarrow bitmap \oplus (0x8000 \gg (r + replicasOfKey))$
20: **end if**
21: $v.slotuse \leftarrow v.slotuse - 1$
22: $v.bitmap \leftarrow bitmap$
23: **return SUCCESS**

**Algorithm 5.6** Insertion in B$^S$-tree

**Require**: key $k$, B$^S$-tree root node $v$

1: compute leaf $v$ and position $r$ by running lines 1-3 of Alg 5.3
2: **if** $v.slotuse < N$ **then**
3: $bitmap \leftarrow \neg v.bitmap$
4: **if** $v.keys[r] == v.keys[r+1]$ **then** ▷ $r$ is an empty slot
5: $v.keys[r] \leftarrow k$
6: $bitmap \leftarrow bitmap \oplus (0x8000 \gg r)$
7: **else**
8: $keysForShift \leftarrow \_lzcnt(bitmap \ll r)$
9: **if** $keysForShift < N$ **then** ▷ empty slot to the right
10: shift right $keysForShift$ keys of node $v$
11: $v.keys[r] \leftarrow k$
12: $bitmap \leftarrow bitmap \oplus (0x8000 \gg (r + keysForShift))$
13: **else** ▷ empty slot to the left
14: $keysForShift \leftarrow \_tzcnt(bitmap \gg (N - r - 1)) - 1$
15: shift left $keysForShift$ keys of node $v$
16: $v.keys[r-1] \leftarrow k$
17: $bitmap \leftarrow bitmap \oplus (0x8000 \gg (r - (keysForShift + 1)))$
18: **end if**
19: $v.slotuse \leftarrow slotuse + 1$
20: $v.bitmap \leftarrow \neg bitmap$
21: **end if**
22: **else**
23: split leaf node $v$
24: **end if**

Table 5.1: Construction time (for 150 million keys)

| **Construction Time (sec)** | | | | | |
|---|---|---|---|---|---|
| **Indices / Datasets** | BOOKS | OSM | FB | GENOME | PLANET |
| $B^S$-tree | **0.33** | 0.33 | 0.33 | 0.33 | 0.33 |
| $CB^S$-tree | 0.35 | **0.32** | **0.18** | **0.20** | **0.18** |
| $B^+$-tree | 0.39 | 0.39 | 0.39 | 0.39 | 0.39 |
| Sparse $B^+$-tree | 0.50 | 0.50 | 0.50 | 0.50 | 0.50 |
| HOT | 15.61 | 16.65 | 16.56 | 16.23 | 15.33 |
| ALEX | 25.43 | 41.60 | 45.46 | 30.74 | 30.06 |
| LIPP | 9.58 | 9.31 | 6.98 | 7.01 | 7.05 |
| DILI | N/R | N/R | 1020 | 973 | 967 |

Table 5.2: Memory footprint (for 150 million keys)

| **Memory Footprint (GB)** | | | | | |
|---|---|---|---|---|---|
| **Indices / Datasets** | BOOKS | OSM | FB | GENOME | PLANET |
| $B^S$-tree | 1.84 | 1.84 | 1.84 | 1.84 | 1.84 |
| $CB^S$-tree | 2.03 | 1.75 | **0.55** | **0.80** | **0.51** |
| $B^+$-tree | **1.41** | **1.41** | 1.41 | 1.41 | 1.41 |
| Sparse-$B^+$-tree | 1.88 | 1.88 | 1.88 | 1.88 | 1.88 |
| HOT | 1.78 | 1.79 | 1.83 | 1.92 | 1.71 |
| ALEX | 2.73 | 2.77 | 2.77 | 2.73 | 2.73 |
| LIPP | 13.51 | 14.69 | 10.89 | 11.66 | 11.62 |
| DILI | N/R | N/R | 8.62 | 9.73 | 7.66 |

Table 5.3: Performance counters for BOOKS, Workload C

| **Workload C - Dataset: BOOKS** | | | | | | |
|---|---|---|---|---|---|---|
| **Index** | Instr. | Cycles | Misp. Branches | L1 Misses | LLC Misses | TLB Misses |
| $B^S$-tree | **204.97** | **836.28** | 1.03 | 16.27 | 16.12 | 0.6 |
| $CB^S$-tree | 364.51 | 1182.3 | **1.02** | 16.59 | 18.6 | **0.06** |
| ALEX | 639.23 | 1211.55 | 5.21 | 21.48 | 23.11 | 2.53 |
| LIPP | 316.62 | 1061.88 | 1.66 | **13.98** | **15.69** | 3.53 |
| HOT | 816.32 | 1838.75 | 3.16 | 27.38 | 32.06 | 3.65 |
| $B^+$-tree | 581.63 | 2534.51 | 11.21 | 33.62 | 38.26 | 4.13 |
| $B^+$-tree-gaps | 539.17 | 2056.22 | 10.13 | 29.72 | 31.76 | 3.21 |

Table 5.4: Performance counters for FB, Workload C

| **Workload C - Dataset: FB** | | | | | | |
|---|---|---|---|---|---|---|
| **Index** | Instr. | Cycles | Misp. Branches | L1 Misses | LLC Misses | TLB Misses |
| $B^S$-tree | **205.27** | **841.82** | **1.03** | 16.32 | 16.16 | 0.6 |
| $CB^S$-tree | 356.05 | 957.96 | 1.07 | 14.4 | **13.42** | **0.00** |
| ALEX | 1013.27 | 2704.73 | 7.06 | 38.21 | 57.67 | 4.17 |
| LIPP | 349.68 | 1197.82 | 1.52 | 16.23 | 17.68 | 4.28 |
| DILI | 326.99 | 1437.36 | 1.87 | **12.41** | 15.4 | 3.37 |
| HOT | 882.93 | 2019.86 | 4.09 | 29.35 | 33.68 | 3.99 |
| $B^+$-tree | 584.45 | 2542.76 | 11.1 | 33.35 | 37.88 | 4.17 |
| $B^+$-tree-gaps | 539.34 | 2047.34 | 10.16 | 28.74 | 30.89 | 3.24 |

# Chapter 6

# Conclusions and future work



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

## 6.1 Summary of Contributions

In this dissertation, we studied parallel and in-memory indexing for temporal, spatial, and relational datasets. Our goal was to design new indexing techniques using modern parallel technologies to improve query efficiency and performance. Another primary objective was to develop simple and effective methods for avoiding duplicate data during partitioning. Avoiding duplicates is crucial as it enhances query performance without requiring complex data structures or inefficient techniques.

**Parallel Partitioning In-Memory for Interval Joins.** In the first part of this thesis, we address the problem of interval joins. We proposed three partitioning strategies: One2One, Temps, and Divs. These strategies use parallelism to improve the performance of interval joins and can enhance both domain-based and hash-based partitioning techniques. We also performed experiments using real datasets to identify the best strategy. The results showed that the Divs strategy is the most efficient and makes the best use of the processor's available cores.

**Parallel In-Memory Evaluation of Spatial Queries.** In the second part of this thesis, we deal with challenges in managing spatial data. First, we focus on the spatial intersection join, which is one of the most time-consuming and resource-intensive queries. A well-known method for handling these queries is the partitioning-based spatial join (PBSM) algorithm. We propose several optimizations to improve the PBSM algorithm and highlight key parameters that greatly affect its efficiency. The first important parameter is the number of partitions, which should be chosen so that the extents of the resulting partitions are approximately one order of magnitude larger than the extents of the rectangles. Another crucial parameter is the choice of the sweeping axis (x or y), as selecting the wrong axis can effectively double the cost of the join. Additionally, we suggest a parallel implementation of the algorithm, which showed excellent scalability in our experiments. This confirms that parallelism significantly improves the performance of spatial joins.

Next, we focus on creating a spatial index optimized for the most common queries, such as range queries and spatial intersection joins. The goal is to design an index that is highly efficient, minimizes the number of comparisons, and avoids generating duplicate results. So, we introduced a secondary partitioning method specifically designed for SOP indices, like grids. This method categorizes MBRs (Minimum Bounding Rectangles) within each spatial partition into four distinct groups. By doing so, it significantly reduces the number of comparisons required for spatial query evaluations and completely avoids generating duplicate results that need to be eliminated later. Additionally, we investigated methods to handle multiple range queries simultaneously, processing them in batches and leveraging parallel execution to improve performance. Our approach is not limited to range queries, as it also effectively eliminates duplicate results during spatial intersection joins. For both range queries and joins, our techniques help avoid redundant computations, saving both time and resources. The experimental results strongly validate our methods. Compared to the best-known duplicate elimination techniques [21], our approach achieves superior performance. Specifically, grids equipped with our method outperform other spatial indices, such as quadtrees and R-trees, by up to one order of magnitude. Finally, spatial intersection join costs were reduced by approximately 50%, thanks to our secondary partitioning strategy and the optimized algorithms we designed for partition-to-partition joins.

**$B^S$-tree: A data-parallel $B^+$-tree for main memory.** In the third part of this thesis, we focus in the optimization of a traditional relational data indexing, $B^+$-tree. we

introduced the $B^S$-tree, an efficient main-memory version of the $B^+$-tree. By incorporating data-parallel branching, innovative gap-handling strategies, and node compression techniques. This work highlights how rethinking indexing structures can address the demands of modern database workloads. We thoroughly compared our $B^S$-tree implementation with open-source versions of leading non-learned and learned indices using popular real-world datasets. The results demonstrate that $B^S$-tree, along with its compressed version ($CB^S$-tree), consistently delivers superior performance across different query and update workloads, achieving throughput that is 1.5 to 2 times higher than the best-performing alternatives from previous research. Our experiments show that, in terms of construction time and memory footprint, $B^S$-tree has a low construction cost, with $B^+$-tree demonstrating comparable performance and a very small memory footprint. Additionally, $CB^S$-tree achieves the fastest construction time and uses 56% to 94% less memory than all other methods on compressible datasets. Finally, performance counter experiments show that our $B^S$-tree and $CB^S$-tree are fully optimized via the use of SIMD instructions, huge pages, and branchless code, aimed at creating an updatable, efficient, and cache-friendly index.

In summary, this dissertation demonstrates how parallel and in-memory indexing can enhance the performance of temporal, spatial, and relational queries. By focusing on efficient partitioning, duplicate avoidance, and innovative data structures, we achieved significant performance improvements across different domains. These contributions highlight the value of combining parallel technologies with straightforward, efficient approaches to address the complexities of modern data management.

## 6.2 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:

**Parallel and Distributed Spatial Data Management Systems.** Numerous distributed and parallel spatial data management systems [16, 17, 115, 182, 183, 184] have been developed, primarily implemented in Java and utilizing traditional spatial indices like quad-trees, R-trees, and R*-trees. Our goal is to integrate our novel 2-layer indexing approach for popular spatial queries into a distributed spatial database system. This system will be implemented in C++ and will use OpenMP and MPI for parallel and distributed processing. It will support various data types, such as points and

rectangles, and handle a wide range of queries, including range, k-NN, and join queries.

**B$^S$-tree future works.** Building on the concepts introduced in [45], we plan to implement a hybrid implementation of B$^S$-tree that maximizes the strengths of both GPUs and CPUs. In this setup, the upper levels of the tree, which are updated less frequently, will be handled by the GPU. This allows us to take full advantage of the GPU's exceptional data parallelism capabilities, enabling faster processing for these less dynamic parts of the structure. The lower levels of the tree, which require frequent updates due to their dynamic nature, will be managed by the CPU. This approach ensures that updates remain fast and efficient while maintaining overall performance balance. Additionally, we plan to incorporate Optimistic Lock Coupling (OLC) [172, 149]. OLC is a technique designed to enable efficient parallel processing of read and write operations, allowing multiple threads to access the data simultaneously without unnecessary delays. This feature will significantly enhance the system's ability to handle workloads that involve a mix of reading and writing operations. Finally, we aim to extend the capabilities of B$^S$-tree by adding support for additional data types, such as strings, which are essential in many applications. We plan to use encoding schemes like Binary, ASCII, or Base64 to store and process string data efficiently. These methods will ensure compatibility with existing systems while maintaining high performance. Together, these improvements will make B$^S$-tree a more powerful, versatile, and efficient data structure for diverse use cases.

# Bibliography


[1] R. Ramakrishnan and J. Gehrke, *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[2] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts, Seventh Edition*. McGraw-Hill Book Company, 2020. [Online]. Available: https://www.db-book.com/

[3] 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*, B. Yormark, Ed. ACM Press, 1984, pp. 47–57. [Online]. Available: https://doi.org/10.1145/602259.602266

[4] Openmp. [Online]. Available: https://www.openmp.org/

[5] Simd. [Online]. Available: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html

[6] 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. [Online]. Available: https://doi.org/10.1007/s00778-003-0111-3

[7] R. H. Güting, "An introduction to spatial database systems," *VLDB J.*, vol. 3, no. 4, pp. 357–399, 1994. [Online]. Available: http://www.vldb.org/journal/VLDBJ3/P357.pdf

[8] P. A. Longley, M. Goodchild, D. J. Maguire, and D. W. Rhind, *Geographic Information Systems and Science*, 3rd ed. Wiley Publishing, 2010.

[9] M. Ester, H. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*

*(KDD-96), Portland, Oregon, USA*, E. Simoudis, J. Han, and U. M. Fayyad, Eds. AAAI Press, 1996, pp. 226–231. [Online]. Available: http://www.aaai.org/Library/KDD/1996/kdd96-037.php

[10] H. Cao, N. Mamoulis, and D. W. Cheung, "Discovery of periodic patterns in spatiotemporal sequences," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 4, pp. 453–467, 2007. [Online]. Available: https://doi.org/10.1109/TKDE.2007.1002

[11] E. H. Jacox and H. Samet, "Spatial join techniques," *ACM Trans. Database Syst.*, vol. 32, no. 1, p. 7, 2007. [Online]. Available: https://doi.org/10.1145/1206049.1206056

[12] P. Bouros and N. Mamoulis, "Spatial joins: what's next?" *ACM SIGSPATIAL Special*, vol. 11, no. 1, pp. 13–21, 2019. [Online]. Available: https://doi.org/10.1145/3355491.3355494

[13] S. Nobari, Q. Qu, and C. S. Jensen, "In-memory spatial join: The data matters!" in *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß, Eds. OpenProceedings.org, 2017, pp. 462–465. [Online]. Available: https://doi.org/10.5441/002/edbt.2017.45

[14] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 1996, pp. 259–270. [Online]. Available: https://doi.org/10.1145/233269.233338

[15] S. Zhang, J. Han, Z. Liu, K. Wang, and Z. Xu, "SJMR: parallelizing spatial join with mapreduce on clusters," in *Proceedings of the 2009 IEEE International Conference on Cluster Computing, August 31 - September 4, 2009, New Orleans, Louisiana, USA*. IEEE Computer Society, 2009, pp. 1–8. [Online]. Available: https://doi.org/10.1109/CLUSTR.2009.5289178

[16] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. H. Saltz, "Hadoop-gis: A high performance spatial data warehousing system over mapreduce," *Proc.*

*VLDB Endow.*, vol. 6, no. 11, pp. 1009–1020, 2013. [Online]. Available: http://www.vldb.org/pvldb/vol6/p1009-aji.pdf

[17] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, J. Gehrke, W. Lehner, K. Shim, S. K. Cha, and G. M. Lohman, Eds. IEEE Computer Society, 2015, pp. 1352–1363. [Online]. Available: https://doi.org/10.1109/ICDE.2015.7113382

[18] T. Brinkhoff, H. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993*, P. Buneman and S. Jajodia, Eds. ACM Press, 1993, pp. 237–246. [Online]. Available: https://doi.org/10.1145/170035.170075

[19] W. G. Aref and H. Samet, "Hashing by proximity to process duplicates in spatial databases," in *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94), Gaithersburg, Maryland, USA, November 29 - December 2, 1994*. ACM, 1994, pp. 347–354. [Online]. Available: https://doi.org/10.1145/191246.191307

[20] V. Pandey, A. Kipf, T. Neumann, and A. Kemper, "How good are modern spatial analytics systems?" *Proc. VLDB Endow.*, vol. 11, no. 11, pp. 1661–1673, 2018. [Online]. Available: http://www.vldb.org/pvldb/vol11/p1661-pandey.pdf

[21] J. Dittrich and B. Seeger, "Data redundancy and duplicate detection in spatial join processing," in *Proceedings of the 16th International Conference on Data Engineering, San Diego, California, USA, February 28 - March 3, 2000*, D. B. Lomet and G. Weikum, Eds. IEEE Computer Society, 2000, pp. 535–546. [Online]. Available: https://doi.org/10.1109/ICDE.2000.839452

[22] N. Mamoulis, *Spatial Data Management*, ser. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011. [Online]. Available: https://doi.org/10.2200/S00394ED1V01Y201111DTM021

[23] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo, "Simba: Efficient in-memory spatial analytics," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June*

*26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1071–1085. [Online]. Available: https://doi.org/10.1145/2882903.2915237

[24] J. Yu, Z. Zhang, and M. Sarwat, "Spatial data management in apache spark: the geospark perspective and beyond," *GeoInformatica*, vol. 23, no. 1, pp. 37–78, 2019. [Online]. Available: https://doi.org/10.1007/s10707-018-0330-9

[25] H. Hoppe, "Progressive meshes," in *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1996, New Orleans, LA, USA, August 4-9, 1996*, J. Fujii, Ed. ACM, 1996, pp. 99–108. [Online]. Available: https://dl.acm.org/citation.cfm?id=237216

[26] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki, "QUASII: query-aware spatial incremental index," in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, Eds. OpenProceedings.org, 2018, pp. 325–336. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.29

[27] C. Cheng, H. Yang, I. King, and M. R. Lyu, "Fused matrix factorization with geographical and social influence in location-based social networks," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*, J. Hoffmann and B. Selman, Eds. AAAI Press, 2012, pp. 17–23. [Online]. Available: https://doi.org/10.1609/aaai.v26i1.8100

[28] M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: scalable incremental processing of continuous queries in spatio-temporal databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. Deßloch, Eds. ACM, 2004, pp. 623–634. [Online]. Available: https://doi.org/10.1145/1007568.1007638

[29] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," *Distributed Parallel Databases*, vol. 15, no. 2, pp. 117–135, 2004. [Online]. Available: https://doi.org/10.1023/B:DAPD.0000013068.25976.88

[30] X. Yu, K. Q. Pu, and N. Koudas, "Monitoring k-nearest neighbor queries over moving objects," in *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, K. Aberer, M. J. Franklin, and S. Nishio, Eds. IEEE Computer Society, 2005, pp. 631–642. [Online]. Available: https://doi.org/10.1109/ICDE.2005.92

[31] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*, F. Özcan, Ed. ACM, 2005, pp. 634–645. [Online]. Available: https://doi.org/10.1145/1066157.1066230

[32] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys, "Trees or grids?: indexing moving objects in main memory," in *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, D. Agrawal, W. G. Aref, C. Lu, M. F. Mokbel, P. Scheuermann, C. Shahabi, and O. Wolfson, Eds. ACM, 2009, pp. 236–245. [Online]. Available: https://doi.org/10.1145/1653771.1653805

[33] S. Ray, R. Blanco, and A. K. Goel, "Supporting location-based services in a main-memory database," in *IEEE 15th International Conference on Mobile Data Management, MDM 2014, Brisbane, Australia, July 14-18, 2014 - Volume 1*, A. B. Zaslavsky, P. K. Chrysanthis, C. Becker, J. Indulska, M. F. Mokbel, D. Nicklas, and C. Chow, Eds. IEEE Computer Society, 2014, pp. 3–12. [Online]. Available: https://doi.org/10.1109/MDM.2014.7

[34] P. Li, H. Lu, Q. Zheng, L. Yang, and G. Pan, "LISA: A learned index structure for spatial data," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 2119–2133. [Online]. Available: https://doi.org/10.1145/3318464.3389703

[35] J. Qi, G. Liu, C. S. Jensen, and L. Kulik, "Effectively learning spatial indices," *Proc. VLDB Endow.*, vol. 13, no. 11, pp. 2341–2354, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p2341-qi.pdf

[36] J. Boyar and K. S. Larsen, "Efficient rebalancing of chromatic search trees," in *Algorithm Theory - SWAT '92, Third Scandinavian Workshop on Algorithm Theory, Helsinki, Finland, July 8-10, 1992, Proceedings*, ser. Lecture Notes in Computer Science, O. Nurmi and E. Ukkonen, Eds., vol. 621. Springer, 1992, pp. 151–164. [Online]. Available: https://doi.org/10.1007/3-540-55706-7_14

[37] T. J. Lehman and M. J. Carey, "A study of index structures for main memory database management systems," in *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, W. W. Chu, G. Gardarin, S. Ohsuga, and Y. Kambayashi, Eds. Morgan Kaufmann, 1986, pp. 294–303. [Online]. Available: http://www.vldb.org/conf/1986/P294.PDF

[38] J. Rao and K. A. Ross, "Cache conscious indexing for decision-support in main memory," in *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, M. P. Atkinson, M. E. Orlowska, P. Valduriez, S. B. Zdonik, and M. L. Brodie, Eds. Morgan Kaufmann, 1999, pp. 78–89. [Online]. Available: http://www.vldb.org/conf/1999/P7.pdf

[39] ——, "Making $b^+$-trees cache conscious in main memory," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, W. Chen, J. F. Naughton, and P. A. Bernstein, Eds. ACM, 2000, pp. 475–486. [Online]. Available: https://doi.org/10.1145/342009.335449

[40] P. Bohannon, P. McIlroy, and R. Rastogi, "Main-memory index structures with fixed-size partial keys," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, pp. 163–174. [Online]. Available: https://doi.org/10.1145/375663.375681

[41] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey, "FAST: fast architecture sensitive tree search on modern cpus and gpus," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA,*

*June 6-10, 2010*, A. K. Elmagarmid and D. Agrawal, Eds. ACM, 2010, pp. 339–350. [Online]. Available: https://doi.org/10.1145/1807167.1807206

[42] J. Fix, A. Wilkes, and K. Skadron, "Accelerating braided b+ tree searches on a gpu with cuda," in *2nd Workshop on Applications for Multi and Many Core Processors: Analysis, Implementation, and Performance (A4MMC), in conjunction with ISCA*. Citeseer, 2011.

[43] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The bw-tree: A b-tree for new hardware platforms," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 302–313. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544834

[44] K. Kaczmarski, "B $^+$ -tree optimized for GPGPU," in *On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part II*, ser. Lecture Notes in Computer Science, R. Meersman, H. Panetto, T. S. Dillon, S. Rinderle-Ma, P. Dadam, X. Zhou, S. Pearson, A. Ferscha, S. Bergamaschi, and I. F. Cruz, Eds., vol. 7566. Springer, 2012, pp. 843–854. [Online]. Available: https://doi.org/10.1007/978-3-642-33615-7_27

[45] A. Shahvarani and H. Jacobsen, "A hybrid b+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1523–1538. [Online]. Available: https://doi.org/10.1145/2882903.2882918

[46] Z. Yan, Y. Lin, L. Peng, and W. Zhang, "Harmonia: a high throughput b+tree for gpus," in *Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019*, J. K. Hollingsworth and I. Keidar, Eds. ACM, 2019, pp. 133–144. [Online]. Available: https://doi.org/10.1145/3293883.3295704

[47] Y. Kwon, S. Lee, Y. Nam, J. C. Na, K. Park, S. K. Cha, and B. Moon, "Db+-tree: A new variant of b+-tree for main-memory database systems," *Inf. Syst.*, vol. 119, p. 102287, 2023. [Online]. Available: https://doi.org/10.1016/j.is.2023.102287

[48] D. R. Morrison, "PATRICIA - practical algorithm to retrieve information coded in alphanumeric," *J. ACM*, vol. 15, no. 4, pp. 514–534, 1968. [Online]. Available: https://doi.org/10.1145/321479.321481

[49] N. Askitis and R. Sinha, "Engineering scalable, cache and space efficient tries for strings," *VLDB J.*, vol. 19, no. 5, pp. 633–660, 2010. [Online]. Available: https://doi.org/10.1007/s00778-010-0183-9

[50] M. Böhm, B. Schlegel, P. B. Volk, U. Fischer, D. Habich, and W. Lehner, "Efficient in-memory indexing with generalized prefix trees," in *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*, ser. LNI, T. Härder, W. Lehner, B. Mitschang, H. Schöning, and H. Schwarz, Eds., vol. P-180. GI, 2011, pp. 227–246. [Online]. Available: https://dl.gi.de/handle/20.500.12116/19581

[51] T. Kissinger, B. Schlegel, D. Habich, and W. Lehner, "*KISS-Tree*: smart latch-free in-memory indexing on modern architectures," in *Proceedings of the Eighth International Workshop on Data Management on New Hardware, DaMoN 2012, Scottsdale, AZ, USA, May 21, 2012*, S. Chen and S. Harizopoulos, Eds. ACM, 2012, pp. 16–23. [Online]. Available: https://doi.org/10.1145/2236584.2236587

[52] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *European Conference on Computer Systems, Proceedings of the Seventh EuroSys Conference 2012, EuroSys '12, Bern, Switzerland, April 10-13, 2012*, P. Felber, F. Bellosa, and H. Bos, Eds. ACM, 2012, pp. 183–196. [Online]. Available: https://doi.org/10.1145/2168836.2168855

[53] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: Artful indexing for main-memory databases," in *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, C. S. Jensen, C. M. Jermaine, and X. Zhou, Eds. IEEE Computer Society, 2013, pp. 38–49. [Online]. Available: https://doi.org/10.1109/ICDE.2013.6544812

[54] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, F. Özcan, G. Koutrika, and S. Madden, Eds. ACM, 2016, pp. 1567–1581. [Online]. Available: https://doi.org/10.1145/2882903.2915222

[55] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis, "HOT: A height optimized trie index for main-memory database systems," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 521–534. [Online]. Available: https://doi.org/10.1145/3183713.3196896

[56] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 323–336. [Online]. Available: https://doi.org/10.1145/3183713.3196931

[57] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 489–504. [Online]. Available: https://doi.org/10.1145/3183713.3196909

[58] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "Fiting-tree: A data-aware index structure," in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 1189–1206. [Online]. Available: https://doi.org/10.1145/3299869.3319860

[59] 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. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1162-ferragina.pdf

[60] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann, D. B. Lomet, and T. Kraska, "ALEX: an updatable adaptive learned index," in *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, Eds. ACM, 2020, pp. 969–984. [Online]. Available: https://doi.org/10.1145/3318464.3389711

[61] J. Zhang and Y. Gao, "CARMI: A cache-aware learned index with a cost-based construction algorithm," *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 2679–2691, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2679-gao.pdf

[62] S. Wu, Y. Cui, J. Yu, X. Sun, T. Kuo, and C. J. Xue, "NFL: robust learned index via distribution transformation," *Proc. VLDB Endow.*, vol. 15, no. 10, pp. 2188–2200, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p2188-wu.pdf

[63] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Radixspline: a single-pass learned index," in *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2020, Portland, Oregon, USA, June 19, 2020*, R. Bordawekar, O. Shmueli, N. Tatbul, and T. K. Ho, Eds. ACM, 2020, pp. 5:1–5:5. [Online]. Available: https://doi.org/10.1145/3401071.3401659

[64] J. Wu, Y. Zhang, S. Chen, Y. Chen, J. Wang, and C. Xing, "Updatable learned index with precise positions," *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1276–1288, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1276-wu.pdf

[65] P. Li, H. Lu, R. Zhu, B. Ding, L. Yang, and G. Pan, "DILI: A distribution-driven learned index," *Proc. VLDB Endow.*, vol. 16, no. 9, pp. 2212–2224, 2023. [Online]. Available: https://www.vldb.org/pvldb/vol16/p2212-li.pdf

[66] S. Zhang, J. Qi, X. Yao, and A. Brinkmann, "Hyper: A high-performance and memory-efficient learned index via hybrid construction," *Proc. ACM*

*Manag. Data*, vol. 2, no. 3, p. 145, 2024. [Online]. Available: https://doi.org/10.1145/3654948

[67] B. Schlegel, R. Gemulla, and W. Lehner, "k-ary search on modern processors," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware, DaMoN 2009, Providence, Rhode Island, USA, June 28, 2009*, P. A. Boncz and K. A. Ross, Eds. ACM, 2009, pp. 52–60. [Online]. Available: https://doi.org/10.1145/1565694.1565705

[68] H. Gunadhi and A. Segev, "Query processing algorithms for temporal intersection joins," in *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*. IEEE Computer Society, 1991, pp. 336–344. [Online]. Available: https://doi.org/10.1109/ICDE.1991.131481

[69] A. Segev and H. Gunadhi, "Event-join optimization in temporal relational databases," in *Proceedings of the Fifteenth International Conference on Very Large Data Bases, August 22-25, 1989, Amsterdam, The Netherlands*, P. M. G. Apers and G. Wiederhold, Eds. Morgan Kaufmann, 1989, pp. 205–215. [Online]. Available: http://www.vldb.org/conf/1989/P205.PDF

[70] J. Enderle, M. Hampel, and T. Seidl, "Joining interval data in relational databases," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. Deßloch, Eds. ACM, 2004, pp. 683–694. [Online]. Available: https://doi.org/10.1145/1007568.1007645

[71] H. Kriegel, M. Pötke, and T. Seidl, "Managing intervals efficiently in object-relational databases," in *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K. Whang, Eds. Morgan Kaufmann, 2000, pp. 407–418. [Online]. Available: http://www.vldb.org/conf/2000/P407.pdf

[72] D. Zhang, V. J. Tsotras, and B. Seeger, "Efficient temporal join processing using indices," in *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, R. Agrawal and K. R.

Dittrich, Eds. IEEE Computer Society, 2002, pp. 103–113. [Online]. Available: https://doi.org/10.1109/ICDE.2002.994701

[73] 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. [Online]. Available: https://doi.org/10.1007/s007780050028

[74] M. D. Soo, R. T. Snodgrass, and C. S. Jensen, "Efficient evaluation of the valid-time natural join," in *Proceedings of the Tenth International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA*. IEEE Computer Society, 1994, pp. 282–292. [Online]. Available: https://doi.org/10.1109/ICDE.1994.283042

[75] I. Sitzmann and P. J. Stuckey, "Improving temporal joins using histograms," in *Database and Expert Systems Applications, 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000, Proceedings*, ser. Lecture Notes in Computer Science, M. T. Ibrahim, J. Küng, and N. Revell, Eds., vol. 1873. Springer, 2000, pp. 488–498. [Online]. Available: https://doi.org/10.1007/3-540-44469-6_46

[76] A. Dignös, M. H. Böhlen, and J. Gamper, "Overlap interval partition join," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds. ACM, 2014, pp. 1459–1470. [Online]. Available: https://doi.org/10.1145/2588555.2612175

[77] F. Cafagna and M. H. Böhlen, "Disjoint interval partitioning," *VLDB J.*, vol. 26, no. 3, pp. 447–466, 2017. [Online]. Available: https://doi.org/10.1007/s00778-017-0456-7

[78] B. Moon, I. F. V. López, and V. Immanuel, "Efficient algorithms for large-scale temporal aggregation," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 744–759, 2003. [Online]. Available: https://doi.org/10.1109/TKDE.2003.1198403

[79] H. Lu, B. C. Ooi, and K. Tan, "On spatially partitioned temporal join," in *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann, 1994, pp. 546–557. [Online]. Available: http://www.vldb.org/conf/1994/P546.PDF

[80] D. Piatov, S. Helmer, and A. Dignös, "An interval join optimized for modern hardware," in *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 1098–1109. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498316

[81] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter, "Scalable sweeping-based spatial join," in *VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, August 24-27, 1998, New York City, New York, USA*, A. Gupta, O. Shmueli, and J. Widom, Eds. Morgan Kaufmann, 1998, pp. 570–581. [Online]. Available: http://www.vldb.org/conf/1998/p570.pdf

[82] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 1173–1184. [Online]. Available: https://doi.org/10.1145/2463676.2465293

[83] P. Bouros and N. Mamoulis, "Interval count semi-joins," in *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, M. H. Böhlen, R. Pichler, N. May, E. Rahm, S. Wu, and K. Hose, Eds. OpenProceedings.org, 2018, pp. 425–428. [Online]. Available: https://doi.org/10.5441/002/edbt.2018.38

[84] M. W. Chekol, G. Pirrò, and H. Stuckenschmidt, "Fast interval joins for temporal SPARQL queries," in *Companion of The 2019 World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, S. Amer-Yahia, M. Mahdian, A. Goel, G. Houben, K. Lerman, J. J. McAuley, R. Baeza-Yates, and L. Zia, Eds. ACM, 2019, pp. 1148–1154. [Online]. Available: https://doi.org/10.1145/3308560.3314997

[85] T. Y. C. Leung and R. R. Muntz, "Temporal query processing and optimization in multiprocessor database machines," in *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, Proceedings*, L. Yuan, Ed. Morgan Kaufmann, 1992, pp. 383–394. [Online]. Available: http://www.vldb.org/conf/1992/P383.PDF

[86] H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz, "Distributed intersection join of complex interval sequences," in *Database Systems for Advanced Applications, 10th International Conference, DASFAA 2005, Beijing, China, April 17-20, 2005, Proceedings*, ser. Lecture Notes in Computer Science, L. Zhou, B. C. Ooi, and X. Meng, Eds., vol. 3453. Springer, 2005, pp. 748–760. [Online]. Available: https://doi.org/10.1007/11408079_68

[87] B. Chawda, H. Gupta, S. Negi, T. A. Faruquie, L. V. Subramaniam, and M. K. Mohania, "Processing interval joins on map-reduce," in *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, S. Amer-Yahia, V. Christophides, A. Kementsietsidis, M. N. Garofalakis, S. Idreos, and V. Leroy, Eds. OpenProceedings.org, 2014, pp. 463–474. [Online]. Available: https://doi.org/10.5441/002/edbt.2014.42

[88] M. Olma, F. Tauheed, T. Heinis, and A. Ailamaki, "BLOCK: efficient execution of spatial range queries in main-memory," in *Proceedings of the 29th International Conference on Scientific and Statistical Database Management, Chicago, IL, USA, June 27-29, 2017*. ACM, 2017, pp. 15:1–15:12. [Online]. Available: https://doi.org/10.1145/3085504.3085519

[89] J. L. Bentley and J. H. Friedman, "Data structures for range searching," *ACM Comput. Surv.*, vol. 11, no. 4, pp. 397–409, 1979. [Online]. Available: https://doi.org/10.1145/356789.356797

[90] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975. [Online]. Available: https://doi.org/10.1145/361002.361007

[91] R. A. Finkel and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, pp. 1–9, 1974. [Online]. Available: https://doi.org/10.1007/BF00288933

[92] M. T. Mahin, T. Hashem, and S. Kabir, "A crowd enabled approach for processing nearest neighbor and range queries in incomplete databases with accuracy guarantee," *Pervasive Mob. Comput.*, vol. 39, pp. 249–266, 2017. [Online]. Available: https://doi.org/10.1016/j.pmcj.2016.09.017

[93] P. Nagarkar, K. S. Candan, and A. Bhat, "Compressed spatial hierarchical bitmap (cshb) indexes for efficiently processing spatial range query workloads," *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1382–1393, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p1382-nagarkar.pdf

[94] D. H. Nguyen, K. Doan, and T. V. Pham, "SIDI: A scalable in-memory density-based index for spatial databases," in *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing, DIDC@HPDC 2016, Kyoto, Japan, June 1, 2016*, E. Yildirim and T. Kosar, Eds. ACM, 2016, pp. 45–52. [Online]. Available: https://doi.org/10.1145/2912152.2912158

[95] H. Samet, *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[96] 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*, H. Garcia-Molina and H. V. Jagadish, Eds. ACM Press, 1990, pp. 322–331. [Online]. Available: https://doi.org/10.1145/93597.98741

[97] D. A. White and R. C. Jain, "Similarity indexing with the ss-tree," in *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, S. Y. W. Su, Ed. IEEE Computer Society, 1996, pp. 516–523. [Online]. Available: https://doi.org/10.1109/ICDE.1996.492202

[98] N. Katayama and S. Satoh, "The sr-tree: An index structure for high-dimensional nearest neighbor queries," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, J. Peckham, Ed. ACM Press, 1997, pp. 369–380. [Online]. Available: https://doi.org/10.1145/253260.253347

[99] K. Kim, S. K. Cha, and K. Kwon, "Optimizing multidimensional index trees for main memory access," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*,

S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, pp. 139–150. [Online]. Available: https://doi.org/10.1145/375663.375679

[100] T. Vu and A. Eldawy, "R*-grove: Balanced spatial partitioning for large-scale datasets," *Frontiers Big Data*, vol. 3, p. 28, 2020. [Online]. Available: https://doi.org/10.3389/fdata.2020.00028

[101] H. Wang, X. Fu, J. Xu, and H. Lu, "Learned index for spatial queries," in *20th IEEE International Conference on Mobile Data Management, MDM 2019, Hong Kong, SAR, China, June 10-13, 2019*. IEEE, 2019, pp. 569–574. [Online]. Available: https://doi.org/10.1109/MDM.2019.00121

[102] F. P. Preparata and M. I. Shamos, *Computational Geometry - An Introduction*, ser. Texts and Monographs in Computer Science. Springer, 1985. [Online]. Available: https://doi.org/10.1007/978-1-4612-1098-6

[103] M. Lo and C. V. Ravishankar, "Spatial hash-joins," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and I. S. Mumick, Eds. ACM Press, 1996, pp. 247–258. [Online]. Available: https://doi.org/10.1145/233269.233337

[104] N. Mamoulis and D. Papadias, "Multiway spatial joins," *ACM Trans. Database Syst.*, vol. 26, no. 4, pp. 424–475, 2001. [Online]. Available: https://doi.org/10.1145/503099.503101

[105] N. Koudas and K. C. Sevcik, "Size separation spatial join," in *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, J. Peckham, Ed. ACM Press, 1997, pp. 324–335. [Online]. Available: https://doi.org/10.1145/253260.253340

[106] M. Pavlovic, F. Tauheed, T. Heinis, and A. Ailamaki, "GIPSY: joining spatial datasets with contrasting density," in *Conference on Scientific and Statistical Database Management, SSDBM '13, Baltimore, MD, USA, July 29 - 31, 2013*, A. Szalay, T. Budavari, M. Balazinska, A. Meliou, and A. Sacan, Eds. ACM, 2013, pp. 11:1–11:12. [Online]. Available: https://doi.org/10.1145/2484838.2484855

[107] M. Pavlovic, T. Heinis, F. Tauheed, P. Karras, and A. Ailamaki, "TRANSFORMERS: robust spatial joins on non-uniform data distributions," in

*32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 673–684. [Online]. Available: https://doi.org/10.1109/ICDE.2016.7498280

[108] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki, "TOUCH: in-memory spatial join by hierarchical data-oriented partitioning," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, K. A. Ross, D. Srivastava, and D. Papadias, Eds. ACM, 2013, pp. 701–712. [Online]. Available: https://doi.org/10.1145/2463676.2463700

[109] S. T. Leutenegger, J. M. Edgington, and M. A. López, "STR: A simple and efficient algorithm for r-tree packing," in *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and P. Larson, Eds. IEEE Computer Society, 1997, pp. 497–506. [Online]. Available: https://doi.org/10.1109/ICDE.1997.582015

[110] F. Tauheed, T. Heinis, and A. Ailamaki, "Configuring spatial grids for efficient main memory joins," in *Data Science - 30th British International Conference on Databases, BICOD 2015, Edinburgh, UK, July 6-8, 2015, Proceedings*, ser. Lecture Notes in Computer Science, S. Maneth, Ed., vol. 9147. Springer, 2015, pp. 199–205. [Online]. Available: https://doi.org/10.1007/978-3-319-20424-6_19

[111] T. Brinkhoff, H. Kriegel, and B. Seeger, "Parallel processing of spatial joins using r-trees," in *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana, USA*, S. Y. W. Su, Ed. IEEE Computer Society, 1996, pp. 258–265. [Online]. Available: https://doi.org/10.1109/ICDE.1996.492114

[112] X. Zhou, D. J. Abel, and D. Truffet, "Data partitioning for parallel spatial join processing," in *Advances in Spatial Databases, 5th International Symposium, SSD'97, Berlin, Germany, July 15-18, 1997, Proceedings*, ser. Lecture Notes in Computer Science, M. Scholl and A. Voisard, Eds., vol. 1262. Springer, 1997, pp. 178–196. [Online]. Available: https://doi.org/10.1007/3-540-63238-7_30

[113] J. M. Patel and D. J. DeWitt, "Clone join and shadow join: two parallel spatial join algorithms," in *ACM-GIS 2000, Proceedings of the Eighth ACM Symposium on Advances in Geographic Information Systems, November 10-11, 2000, Washington D.C., USA*, K. Li, K. Makki, N. Pissinou, and S. Ravada, Eds. ACM, 2000, pp. 54–61. [Online]. Available: https://doi.org/10.1145/355274.355282

[114] A. Cary, Z. Sun, V. Hristidis, and N. Rishe, "Experiences on processing spatial data with mapreduce," in *Scientific and Statistical Database Management, 21st International Conference, SSDBM 2009, New Orleans, LA, USA, June 2-4, 2009, Proceedings*, ser. Lecture Notes in Computer Science, M. Winslett, Ed., vol. 5566. Springer, 2009, pp. 302–319. [Online]. Available: https://doi.org/10.1007/978-3-642-02279-1_24

[115] S. You, J. Zhang, and L. Gruenwald, "Large-scale spatial join query processing in cloud," in *31st IEEE International Conference on Data Engineering Workshops, ICDE Workshops 2015, Seoul, South Korea, April 13-17, 2015*. IEEE Computer Society, 2015, pp. 34–41. [Online]. Available: https://doi.org/10.1109/ICDEW.2015.7129541

[116] M. M. Alam, S. Ray, and V. C. Bhavsar, "A performance study of big spatial data systems," in *Proceedings of the 7th ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2018, Seattle, WA, USA, November 6, 2018*, V. Chandola and R. R. Vatsavai, Eds. ACM, 2018, pp. 1–9. [Online]. Available: https://doi.org/10.1145/3282834.3282841

[117] G. Graefe, "Modern b-tree techniques," *Found. Trends Databases*, vol. 3, no. 4, pp. 203–402, 2011. [Online]. Available: https://doi.org/10.1561/1900000028

[118] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Benjamin/Cummings, 1989.

[119] R. Bayer and E. M. McCreight, "Organization and maintenance of large ordered indexes," in *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access, November 15-16, 1970, Rice University, Houston, Texas, USA (Second Edition with an Appendix)*, E. F. Codd, Ed. ACM, 1970, pp. 107–141. [Online]. Available: https://doi.org/10.1145/1734663.1734671

[120] D. Comer, "The ubiquitous b-tree," *ACM Comput. Surv.*, vol. 11, no. 2, pp. 121–137, 1979. [Online]. Available: https://doi.org/10.1145/356770.356776

[121] R. Bayer, "Symmetric binary b-trees: Data structure and maintenance algorithms," *Acta Informatica*, vol. 1, pp. 290–306, 1972. [Online]. Available: https://doi.org/10.1007/BF00289509

[122] L. J. Guibas and R. Sedgewick, "A dichromatic framework for balanced trees," in *19th Annual Symposium on Foundations of Computer Science, Ann Arbor, Michigan, USA, 16-18 October 1978*. IEEE Computer Society, 1978, pp. 8–21. [Online]. Available: https://doi.org/10.1109/SFCS.1978.3

[123] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996. [Online]. Available: https://doi.org/10.1007/s002360050048

[124] H. Zhang, G. Chen, B. C. Ooi, K. Tan, and M. Zhang, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, no. 7, pp. 1920–1948, 2015. [Online]. Available: https://doi.org/10.1109/TKDE.2015.2427795

[125] S. Chen, P. B. Gibbons, and T. C. Mowry, "Improving index performance through prefetching," in *Proceedings of the 2001 ACM SIGMOD international conference on Management of data, Santa Barbara, CA, USA, May 21-24, 2001*, S. Mehrotra and T. K. Sellis, Eds. ACM, 2001, pp. 235–246. [Online]. Available: https://doi.org/10.1145/375663.375688

[126] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching b$^+$trees: optimizing both cache and disk performance," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM, 2002, pp. 157–168. [Online]. Available: https://doi.org/10.1145/564691.564710

[127] R. A. Hankins and J. M. Patel, "Effect of node size on the performance of cache-conscious b$^+$-trees," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2003, June 9-14, 2003, San Diego, CA, USA*, B. Cheng, S. K. Tripathi, J. Rexford,

and W. H. Sanders, Eds. ACM, 2003, pp. 283–294. [Online]. Available: https://doi.org/10.1145/781027.781063

[128] J. Zhou and K. A. Ross, "Buffering accesses to memory-resident index structures," in *Proceedings of 29th International Conference on Very Large Data Bases, VLDB 2003, Berlin, Germany, September 9-12, 2003*, J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Morgan Kaufmann, 2003, pp. 405–416. [Online]. Available: http://www.vldb.org/conf/2003/papers/S13P02.pdf

[129] G. Graefe and P. Larson, "B-tree indexes and CPU caches," in *Proceedings of the 17th International Conference on Data Engineering, April 2-6, 2001, Heidelberg, Germany*, D. Georgakopoulos and A. Buchmann, Eds. IEEE Computer Society, 2001, pp. 349–358. [Online]. Available: https://doi.org/10.1109/ICDE.2001.914847

[130] G. Graefe, "B-tree indexes, interpolation search, and skew," in *Workshop on Data Management on New Hardware, DaMoN 2006, Chicago, Illinois, USA, June 25, 2006*, A. Ailamaki, P. A. Boncz, and S. Manegold, Eds. ACM, 2006, p. 5. [Online]. Available: https://doi.org/10.1145/1140402.1140409

[131] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, USA, June 3-6, 2002*, M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM, 2002, pp. 145–156. [Online]. Available: https://doi.org/10.1145/564691.564709

[132] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors," *Proc. VLDB Endow.*, vol. 4, no. 11, pp. 795–806, 2011. [Online]. Available: http://www.vldb.org/pvldb/vol4/p795-sewall.pdf

[133] C. Wu, T. Kuo, and L. Chang, "An efficient b-tree layer implementation for flash-memory storage systems," *ACM Trans. Embed. Comput. Syst.*, vol. 6, no. 3, p. 19, 2007. [Online]. Available: https://doi.org/10.1145/1275986.1275991

[134] D. Agrawal, D. Ganesan, R. K. Sitaraman, Y. Diao, and S. Singh, "Lazy-adaptive tree: An optimized index structure for flash devices," *Proc.*

*VLDB Endow.*, vol. 2, no. 1, pp. 361–372, 2009. [Online]. Available: http://www.vldb.org/pvldb/vol2/vldb09-341.pdf

[135] Y. Li, B. He, J. Yang, Q. Luo, and K. Yi, "Tree indexing on solid state drives," *Proc. VLDB Endow.*, vol. 3, no. 1, pp. 1195–1206, 2010. [Online]. Available: http://www.vldb.org/pvldb/vldb2010/pvldb_vol3/R106.pdf

[136] M. V. Jørgensen, R. B. Rasmussen, S. Saltenis, and C. Schjønning, "Fb-tree: a $b^+$-tree for flash-based ssds," in *15th International Database Engineering and Applications Symposium (IDEAS 2011), September 21 - 27, 2011, Lisbon, Portugal*, B. C. Desai, I. F. Cruz, and J. Bernardino, Eds. ACM, 2011, pp. 34–42. [Online]. Available: https://doi.org/10.1145/2076623.2076629

[137] G. Na, S. Lee, and B. Moon, "Dynamic in-page logging for $b^+$-tree index," *IEEE Trans. Knowl. Data Eng.*, vol. 24, no. 7, pp. 1231–1243, 2012. [Online]. Available: https://doi.org/10.1109/TKDE.2011.32

[138] M. Athanassoulis and A. Ailamaki, "Bf-tree: Approximate tree indexing," *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1881–1892, 2014. [Online]. Available: http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf

[139] P. Jin, C. Yang, C. S. Jensen, P. Yang, and L. Yue, "Read/write-optimized tree indexing for solid-state drives," *VLDB J.*, vol. 25, no. 5, pp. 695–717, 2016. [Online]. Available: https://doi.org/10.1007/s00778-015-0406-1

[140] L. Wang, Z. Zhang, B. He, and Z. Zhang, "Pa-tree: Polled-mode asynchronous B+ tree for nvme," in *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*. IEEE, 2020, pp. 553–564. [Online]. Available: https://doi.org/10.1109/ICDE48307.2020.00054

[141] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proc. VLDB Endow.*, vol. 8, no. 7, pp. 786–797, 2015. [Online]. Available: http://www.vldb.org/pvldb/vol8/p786-chen.pdf

[142] J. Liu, S. Chen, and L. Wang, "Lb+-trees: Optimizing persistent index performance on 3dxpoint memory," *Proc. VLDB Endow.*, vol. 13, no. 7, pp. 1078–1090, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol13/p1078-liu.pdf

[143] D. Siakavaras, P. Billis, K. Nikas, G. I. Goumas, and N. Koziris, "Efficient concurrent range queries in b+-trees using RCU-HTM," in *SPAA '20: 32nd ACM Symposium on Parallelism in Algorithms and Architectures, Virtual Event, USA, July 15-17, 2020*, C. Scheideler and M. Spear, Eds. ACM, 2020, pp. 571–573. [Online]. Available: https://doi.org/10.1145/3350755.3400237

[144] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, 1960. [Online]. Available: https://doi.org/10.1145/367390.367400

[145] D. E. Knuth, *The art of computer programming, , Volume III, 2nd Edition*. Addison-Wesley, 1998. [Online]. Available: https://www.worldcat.org/oclc/312994415

[146] R. D. L. Briandais, "File searching using variable length keys," in *Papers presented at the the 1959 western joint computer conference, IRE-AIEE-ACM 1959 (Western), San Francisco, California, USA, March 3-5, 1959*, R. R. Johnson, Ed. ACM, 1959, pp. 295–298. [Online]. Available: https://doi.org/10.1145/1457838.1457895

[147] N. Askitis and R. Sinha, "Hat-trie: A cache-conscious trie-based data structure for strings," in *Computer Science 2007. Proceedings of the Thirtieth Australasian Computer Science Conference (ACSC2007). Ballarat, Victoria, Australia, January 30 - February 2, 2007. Proceedings*, ser. CRPIT, G. Dobbie, Ed., vol. 62. Australian Computer Society, 2007, pp. 97–105. [Online]. Available: http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV62Askitis.html

[148] N. Askitis and J. Zobel, "B-tries for disk-based string management," *VLDB J.*, vol. 18, no. 1, pp. 157–179, 2009. [Online]. Available: https://doi.org/10.1007/s00778-008-0094-1

[149] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016*. ACM, 2016, pp. 3:1–3:8. [Online]. Available: https://doi.org/10.1145/2933349.2933352

[150] C. Tang, Y. Wang, Z. Dong, G. Hu, Z. Wang, M. Wang, and H. Chen, "Xindex: a scalable learned index for multicore data storage," in

*PPoPP '20: 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California, USA, February 22-26, 2020*, R. Gupta and X. Shen, Eds. ACM, 2020, pp. 308–320. [Online]. Available: https://doi.org/10.1145/3332466.3374547

[151] P. Li, Y. Hua, J. Jia, and P. Zuo, "Finedex: A fine-grained learned index scheme for scalable and concurrent memory systems," *Proc. VLDB Endow.*, vol. 15, no. 2, pp. 321–334, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p321-hua.pdf

[152] Y. Wang, C. Tang, Z. Wang, and H. Chen, "Sindex: a scalable learned index for string keys," in *APSys '20: 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, August 24-25, 2020*, T. Kim and P. P. C. Lee, Eds. ACM, 2020, pp. 17–24. [Online]. Available: https://doi.org/10.1145/3409963.3410496

[153] J. Ge, H. Zhang, B. Shi, Y. Luo, Y. Guo, Y. Chai, Y. Chen, and A. Pan, "SALI: A scalable adaptive learned index framework based on probability models," *Proc. ACM Manag. Data*, vol. 1, no. 4, pp. 258:1–258:25, 2023. [Online]. Available: https://doi.org/10.1145/3626752

[154] M. Mitzenmacher, "A model for learned bloom filters and optimizing by sandwiching," in *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds., 2018, pp. 462–471. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/0f49c89d1e7298bb9930789c8ed59d48-Abstract.html

[155] B. Lu, J. Ding, E. Lo, U. F. Minhas, and T. Wang, "APEX: A high-performance learned index on persistent memory," *Proc. VLDB Endow.*, vol. 15, no. 3, pp. 597–610, 2021. [Online]. Available: http://www.vldb.org/pvldb/vol15/p597-lu.pdf

[156] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska, "Benchmarking learned indexes," *Proc. VLDB Endow.*, vol. 14, no. 1, pp. 1–13, 2020. [Online]. Available: http://www.vldb.org/pvldb/vol14/p1-marcus.pdf

[157] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "Sosd: A benchmark for learned indexes," *NeurIPS Workshop on Machine Learning for Systems*, 2019.

[158] C. Wongkham, B. Lu, C. Liu, Z. Zhong, E. Lo, and T. Wang, "Are updatable learned indexes ready?" *Proc. VLDB Endow.*, vol. 15, no. 11, pp. 3004–3017, 2022. [Online]. Available: https://www.vldb.org/pvldb/vol15/p3004-wongkham.pdf

[159] 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. [Online]. Available: http://www.vldb.org/pvldb/vol10/p1346-bouros.pdf

[160] 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 *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*. IEEE, 2014, pp. 511–516. [Online]. Available: https://doi.org/10.1109/SmartGridComm.2014.7007698

[161] L. Isella, J. Stehlé, A. Barrat, C. Cattuto, J.-F. Pinton, and W. V. den Broeck, "What's in a crowd? analysis of face-to-face behavioral networks," *Journal of Theoretical Biology*, vol. 271, no. 1, pp. 166–180, 2011.

[162] G. P. Copeland and S. Khoshafian, "A decomposition storage model," in *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, USA, May 28-31, 1985*, S. B. Navathe, Ed. ACM Press, 1985, pp. 268–279. [Online]. Available: https://doi.org/10.1145/318898.318923

[163] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik, "C-store: A column-oriented DBMS," in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, Eds. ACM, 2005, pp. 553–564. [Online]. Available: http://www.vldb.org/archives/website/2005/program/paper/thu/p553-stonebraker.pdf

[164] D. Tsitsigkos, P. Bouros, N. Mamoulis, and M. Terrovitis, "Parallel in-memory evaluation of spatial joins," in *Proceedings of the 27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2019, Chicago, IL, USA, November 5-8, 2019*, F. B. Kashani, G. Trajcevski, R. H. Güting, L. Kulik, and S. D. Newsam, Eds. ACM, 2019, pp. 516–519. [Online]. Available: https://doi.org/10.1145/3347146.3359343

[165] I. Sabek and M. F. Mokbel, "On spatial joins in mapreduce," in *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2017, Redondo Beach, CA, USA, November 7-10, 2017*, E. G. Hoel, S. D. Newsam, S. Ravada, R. Tamassia, and G. Trajcevski, Eds. ACM, 2017, pp. 21:1–21:10. [Online]. Available: https://doi.org/10.1145/3139958.3139967

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

[167] G. Kedem, "The quad-cif tree: A data structure for hierarchical on-line algorithms," in *Proceedings of the 19th Design Automation Conference, DAC '82, Las Vegas, Nevada, USA, June 14-16, 1982*, J. S. Crabbe, C. E. Radke, and H. Ofek, Eds. ACM/IEEE, 1982, pp. 352–357. [Online]. Available: https://doi.org/10.1145/800263.809229

[168] R. Bayer and M. Schkolnick, "Concurrency of operations on b-trees," *Acta Informatica*, vol. 9, pp. 1–21, 1977. [Online]. Available: https://doi.org/10.1007/BF00263762

[169] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 650–670, 1981. [Online]. Available: https://doi.org/10.1145/319628.319663

[170] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, R. Govindarajan, D. A. Padua, and M. W. Hall, Eds. ACM, 2010, pp. 257–268. [Online]. Available: https://doi.org/10.1145/1693453.1693488

[171] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen, "Building a bw-tree takes more than just buzz words," in *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, G. Das, C. M. Jermaine, and P. A. Bernstein, Eds. ACM, 2018, pp. 473–488. [Online]. Available: https://doi.org/10.1145/3183713.3196895

[172] V. Leis, M. Haubenschild, and T. Neumann, "Optimistic lock coupling: A scalable and efficient general-purpose synchronization method," *IEEE Data Eng. Bull.*, vol. 42, no. 1, pp. 73–84, 2019. [Online]. Available: http://sites.computer.org/debull/A19mar/p73.pdf

[173] P. V. Sandt, Y. Chronis, and J. M. Patel, "Efficiently searching in-memory sorted arrays: Revenge of the interpolation search?" in *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, Eds. ACM, 2019, pp. 36–53. [Online]. Available: https://doi.org/10.1145/3299869.3300075

[174] S. S. Rao, M. H. Huntley, N. C. Durand, E. K. Stamenova, I. D. Bochkov, J. T. Robinson, A. L. Sanborn, I. Machol, A. D. Omer, E. S. Lander *et al.*, "A 3d map of the human genome at kilobase resolution reveals principles of chromatin looping," *Cell*, vol. 159, no. 7, pp. 1665–1680, 2014.

[175] (2017) Google cloud. openstreetmap. [Online]. Available: https://console.cloud.google.com/marketplace/details/openstreetmap/geo-openstreetmap?project=practice-bigtable.

[176] (2013) Stx b+tree code. [Online]. Available: https://github.com/bingmann/stx-btree/tree/master

[177] (2018) Hot code. [Online]. Available: https://github.com/speedskater/hot

[178] (2020) Alex code. [Online]. Available: https://github.com/microsoft/ALEX

[179] (2021) Lipp code. [Online]. Available: https://github.com/Jiacheng-WU/lipp

[180] (2023) Dili code. [Online]. Available: https://github.com/pfl-cs/DILI

[181] (2018) Perfevent code. [Online]. Available: https://github.com/viktorleis/perfevent?tab=readme-ov-file

[182] Magellan: Geospatial analytics using spark. [Online]. Available: https://github.com/harsha2010/magellan

[183] D. Xie, F. Li, B. Yao, G. Li, Z. Chen, L. Zhou, and M. Guo, "Simba: spatial in-memory big data analysis," in *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, S. Ravada, M. E. Ali, S. D. Newsam, M. Renz, and G. Trajcevski, Eds. ACM, 2016, pp. 86:1–86:4. [Online]. Available: https://doi.org/10.1145/2996913.2996935

[184] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *Proc. VLDB Endow.*, vol. 9, no. 13, pp. 1565–1568, 2016. [Online]. Available: http://www.vldb.org/pvldb/vol9/p1565-tang.pdf

# Author's Publications

## Main Publications

- Dimitris Tsitsigkos, Achilleas Michalopoulos, Nikos Mamoulis, Manolis Terrovitis, **B$^S$-tree: A data-parallel B+-tree for main memory**, submitted.

- Dimitris Tsitsigkos, Panagiotis Bouros, Konstantinos Lampropoulos, Nikos Mamoulis, Manolis Terrovitis, **Two-layer Space-oriented Partitioning for Non-point Data** in *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 36(3): 1341-1355, March 2024.

- Dimitris Tsitsigkos, Konstantinos Lampropoulos, Panagiotis Bouros, Nikos Mamoulis, Manolis Terrovitis, **A Two-layer Partitioning for Non-point Spatial Data** in *37th International Conference on Data Engineering (ICDE)*, Chania, Greece, April 2021.

- Dimitris Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, Manolis Terrovitis, **Parallel In-Memory Evaluation of Spatial Joins** in *27th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (SIGSPATIAL/GIS)*, Chicago, IL, November, 2019 (short paper).

## Other Publications

- Achilleas Michalopoulos, Dimitris Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, Manolis Terrovitis, **Efficient Distance Queries on Non-point Data**, in *ACM Transactions on Spatial Algorithms and Systems (ACM TSAS)*, Vol 11, No 1, March 2025.

- Achilleas Michalopoulos, Dimitris Tsitsigkos, Panagiotis Bouros, Nikos Mamoulis, Manolis Terrovitis, **Efficient Nearest Neighbor Queries on Non-point Data** in

Proceedings of the *International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL)*, Hamburg, Germany, November 2023 (short paper).

- Panagiotis Bouros, Nikos Mamoulis, Dimitris Tsitsigkos, Manolis Terrovitis, **In-Memory Interval Joins**, in *Very Large Data Base Journal (VLDBJ)*, 30(4): 667-691, July 2021.

- Panagiotis Bouros, Konstantinos Lampropoulos, Dimitris Tsitsigkos, Nikos Mamoulis, Manolis Terrovitis, **Band Joins for Interval Data** in *23rd International Conference on Extending Database Technology (EDBT)*, Copenhagen, Denmark, March 2020 (short paper).

# Short Biography

Dimitris Tsitsigkos was born in 1989. He holds a B.Sc. in Informatics and Telecommunications (2012) and an M.Sc. in Computing Systems: Software and Hardware (2016) from the National and Kapodistrian University of Athens. Since 2012, he has been a software engineer at the Institute for the Management of Information Systems (IMIS) at the Athena Research Center, supervised by researcher Manolis Terrovitis. During his Ph.D., he interned at the University of Mainz in Germany with Professor Panagiotis Bouros. His research focuses on data management, query processing, parallel algorithms, real-time analytics, and data anonymization.