Efficient Query Processing for Spatial and Temporal Data Exploration

Thèse N° 9637

Présentée le 9 août 2019

à la Faculté informatique et communications
Laboratoire de systèmes et applications de traitement de données massives
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Eleni TZIRITA ZACHARATOU

Acceptée sur proposition du jury

Prof. A. Argyraki, présidente du jury
Prof. A. Ailamaki, directrice de thèse
Prof. J. Freire, rapporteuse
Prof. Y. Ioannidis, rapporteur
Prof. K. Aberer, rapporteur

2019

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

In order to seek truth,
it is necessary once in the course of our life,
to doubt, as far as possible, of all things.
— René Descartes

To my parents, Giorgos and Aggeliki,
my sisters, Tonia and Nikoleta,
and my partner, Mickaël.

Acknowledgements

Foremost, I would like to thank my advisor, *Prof. Anastasia Ailamaki*, for her faith in me, for her advice on research, career, and life in general, and for creating an amazing research environment. Her strength, persistence and endless energy have been, and will continue to be, inspiring.

I would also like to thank *Prof. Juliana Freire* for a great internship and research experience, for inspiring me to explore new research directions and helping me shape a significant part of my thesis, and for being in my thesis committee.

I am also very grateful to *Prof. Yannis Ioannidis* and *Prof. Karl Aberer* for serving in my thesis committee and for their invaluable feedback on my work. I want to also thank *Prof. Katerina Argyraki* for giving generously her time to preside over my thesis committee.

I was lucky to have *Prof. Thomas Heinis* and *Dr. Harish Doraiswamy* as my mentors. This thesis would not have been possible without their contribution. I would also like to thank *Dr. Darius Šidlauskas* and *Dr. Farhan Tauheed* for the fruitful collaboration.

A big thanks goes to the *DIAS lab* family. I had the privilege of being surrounded by a great group of people: *Adrian, Angelos, Aunn, Ben, Cesar, Christina, Danica, Darius, Diane, Dimitra, Erietta, Erika, Farhan, Foteini, Giorgos, Iraklis, Kostas, Lionel, Manos (×2), Matt, Miguel, Mira, Odysseas, Panos, Periklis, Pinar, Radu, Raja, Rakesh, Renata, Satya, Sharareh, Snow, Stella, Tahir, Utku, Victor*, and *Yannis*. I thank you all for being amazing colleagues, and for always providing thoughtful feedback and asking challenging questions. *Giorgos* has been the toughest critic of my writings, helping me to significantly improve the quality of my text, this thesis included. Thanks to *Christina* for making life on campus colorful by discovering all the fun events around–especially those with free food. Thanks to *Matt*, for being Matt, for his generosity, and his styling and social media advice. *Mira* and I shared the same PhD journey. She has been my office mate since my very first day in the lab and my main brainstorm buddy. Seeing her succeed gave me confidence that success was possible. Mira is much more than a colleague, she is also a caring friend with whom I had great time outside the lab. Finally, special thanks to *Erika* and *Dimitra* for their indispensable help with administrative issues.

I am fortunate enough to have a second "lab family" overseas, the *VIDA lab* at NYU. Thank you all for making my internship an unforgettable experience.

Acknowledgements

Many friends supported me in this journey, and made life so much more fun. *Florin* provided great feedback on this document, along with invaluable research, career, and traveling advice. Ever since I met *Georgia*–remember our CrimeShield project?–she brought a lot of joy to my life at and around EPFL. She has been a lunch and coffee buddy, a beer buddy, and a party buddy in general, also serving as a personal photographer. Thank you for your friendship! Thanks to *Adrian*, *Daniel* and *Immanuel* for taking me out for dinner when I was a newcomer in Lausanne, and for giving me PhD advice. Thanks to the "GOT gang", *João, David*, and *Emeline*. Although the last season of GOT was bad, watching it with you made me relax when I needed it the most–in the last month of writing this thesis. Thanks to the "Dinner Time gang"–*Julie, Glenn, Solène, Chris, Morgane, Jeremy, Loïc, Vishal*, and *Jean-Fabien*–for all the delicious and fun dinners. I also want to thank my long-term friends from NTUA, *Konstantina*, *Erie*, *Argyro*, *Eleni* and *Fivi*. Thanks for always keeping in touch! Finally, special thanks to *Nikos* for his care and support. Nikos has been instrumental in my decision to study computer science, and always gives me the best coding advice.

Mickaël made Lausanne home for me. Having him by my side made this thesis possible. Thank you for your love, for helping me remain calm in times of failure, and supporting me in any way you could. Thanks are also due for your delicious cooking, and for making the best brownies for my private thesis defense. I am looking forward to our future adventures!

Last, but definitely not least, I would like to thank my family–my father, *Giorgos*, my mother, *Aggeliki*, and my two sisters, *Tonia* and *Nikoleta*–for their continuous support, guidance, advice and encouragement throughout my studies. Thank you for your love, for believing in me, and for always inspiring me to be a better person.

This research has been supported by grants from the School of Computer and Communication Sciences, EPFL, the European Union Seventh Framework Programme Grants no. 604102 (FP7/2007-2013 - HBP) and 617508 (ERC-2013-CoG – ViDa), the European Union's Horizon 2020 research and innovation programme under grant agreements no. 650003 and 720270 (Human Brain project), and the Moore-Sloan Data Science Environment at NYU.

Lausanne, 10 July 2019 Eleni Tzirita Zacharatou

Abstract

Core to many scientific and analytics applications are spatial data capturing the position or shape of objects in space, and time series recording the values of a process over time. Effective analysis of such data requires a shift from confirmatory pipelines to exploratory ones. However, there is a mismatch between the requirements of spatial and temporal data exploration and the capabilities of the data management solutions available today. First, traditional spatial query operators evaluate spatial relations with time-consuming geometric tests that oppose the interactivity expected from exploratory applications, creating an undue overhead. Second, spatial access methods are built on top of rough geometric object approximations that do not capture the complex structure and distribution of today's spatial data and are thus inefficient. Third, traditional indexes are typically built upfront before queries can be processed and over single data attributes, thus precluding interactive accesses to interesting data subsets that may be specified with constraints on multiple attributes. Finally, existing access methods scale poorly with increasingly granular spatial and temporal data originating from ever more precise data acquisition technologies and ever faster computing infrastructure.

This thesis introduces a novel family of spatial and temporal access methods and query operators that aim to bridge the gap between existing data management techniques and data exploration applications. We show that spatial query operators can be decomposed into primitive graphics operations that are efficiently executed by graphics hardware (GPU) and allow to trade accuracy for interactivity. Furthermore, we design access methods that adapt to data characteristics, data growth trends, and workload access patterns, thereby providing scalable performance for ad-hoc queries over increasing data amounts. Specifically, we introduce a spatial approximation that adapts to the structural properties and distribution of the data, and propose spatial and time series access methods that leverage similarities between data items and support filtering over multiple attributes. Finally, we present an approach that indexes data incrementally, using queries as hints for optimizing data access.

Keywords: data management, database management systems, scientific data management, visual analytics systems, data exploration, spatial data management, spatial data analytics, temporal data management, multidimensional data access methods, time series access methods, geospatial joins, GPU rasterization, spatial approximation, incremental indexing, bitmap indexing, quadtree compression

Résumé

Au centre de nombreuses applications scientifiques et analytiques sont des données spatiales qui capturent la position ou la forme d'objets dans l'espace, et de séries temporelles qui contiennent les valeurs d'un processus évoluant dans le temps. Analyser efficacement ces données requiert des chaînes de traitement de passer d'un mode confirmatoire à un mode exploratoire. Il y a cependant un décalage entre les exigences requises par l'exploration de données spatiales et temporelles et les capacités des solutions de gestion de données disponibles de nos jours. Premièrement, les opérateurs de requêtes spatiaux traditionnels évaluent les relations spatiales avec des tests géométriques coûteux en temps qui s'opposent à l'interactivité désirée des applications exploratoires en induisant un coût additionnel injustifié. Deuxièmement, les méthodes d'accès spatiales sont basées sur des approximations grossières de la géométrie des objets qui ne capturent pas la structure et distribution complexes des données spatiales d'aujourd'hui et qui sont donc inefficaces. Troisièmement, les index sont traditionnellement construits avant que les requêtes ne soient exécutées et sur un unique attribut des données, excluant un accès interactif aux sous-ensembles d'intérêts des données, éventuellement spécifiés par des contraintes sur des attributs multiples. Finalement, les méthodes d'accès existantes se mettent difficilement à l'échelle de la granularité croissante des données spatiales et temporelles provenant de technologies d'acquisition de données de plus en plus précises et d'infrastructures de calculs de plus en plus rapides.

Cette thèse introduit une nouvelle famille de méthodes d'accès aux données spatiales et temporelles et des opérateurs de requêtes qui ont pour objectif de combler l'écart entre les techniques de gestion de données existantes et les applications d'exploration de données. Nous démontrons que les opérateurs de requêtes spatiales peuvent être décomposés en opérations graphiques primitives qui sont efficacement exécutées par des processeurs graphiques (GPU) qui rendent possible l'interactivité en sacrifiant une part de précision. De plus, nous concevons des méthodes d'accès qui s'adaptent aux caractéristiques des données, à leurs volumes croissants et aux schémas d'accès des charges de travail, ce qui fournit aux requêtes ad-hoc des performances à l'échelle des quantités grandissantes de données. Plus précisément, nous introduisons une approximation spatiale qui s'adapte aux propriétés structurelles et à la distribution des données, et proposons des méthodes d'accès aux données spatiales et aux séries temporelles qui s'appuient sur les similarités entre les éléments de données tout en supportant le filtrage sur de multiples attributs. Finalement, nous présentons une approche qui indexe incrémentalement les données, en utilisant les requêtes comme indices pour en optimiser l'accès.

Mots-clés : gestion de données, systèmes de gestion de base de données, gestion de données scientifiques, systèmes d'analyse visuelle, exploration de données, gestion de données spatiales, analyse de données spatiales, gestion de données temporelles, méthodes d'accès aux données multidimensionnelles, méthodes d'accès aux séries temporelles, jointures géospatiales, rastérisation (GPU), approximation spatiale, indexation incrémentielle, indexation de bitmaps, compression d'arbres quaternaires

Contents

List of Figures

List of Tables

1 Introduction

There is an explosion in the amount of spatial and temporal data being generated and collected today. Billions of mobile devices, cars, social networks, satellites, sensors, scientific simulations, and many other sources produce spatial and temporal data constantly. Uber, a popular Transportation Network Company (TNC), recently announced hitting 10 billion rides [151], doubling the number of completed rides in roughly one year [152]. Furthermore, location-based services are growing in popularity, generating a large amount of location data on a daily basis. Foursquare, a mobile application that allows users to "check in" to places that they visit and provide recommendations of local venues, contains over 105 million venues around the world and has more than 12 billion check-ins to date [53]. Twitter generates approximately 10 million geo-tagged tweets every single day [57]. City administrations all over the world are collecting and making available increasing volumes of urban data containing spatial and temporal information, such as transportation (taxi trips, bus, subway), reported crimes, and land-use data [113]. Scientists in the Human Brain Project [73] build three-dimensional models of brain structures, and simulate them over time, producing time series that capture brain activity. The ultimate goal is to simulate the human brain that contains 86 billion neurons.

The value of this wealth of data is tied to its analysis. Analyzing the large amounts of spatial and temporal data can lead to a variety of new discoveries, better services, and new products. Uber uses data visualizations to generate insights on how to provide better service and make city transportation more efficient [154]. The analysis of geo-tagged tweets can play an important role in handling disasters by indicating locations that need food or medical supplies [111]. Domain experts and city governments can make more informed policies and planning decisions by exploring urban data [51]. In scientific simulations, analyzing structural and temporal information allows to refine the models and re-calibrate the simulation parameters [106]. To empower domain experts (without a data science background) to perform such analyses effectively, there is a paradigm shift from batch-oriented analysis pipelines to exploratory ones.

Effective data exploration requires interactivity [99] and support for ad-hoc queries, ranging from aggregations at different levels of resolution to simultaneous filtering over different data attributes.

Data management systems, however, face many challenges in supporting exploratory pipelines for spatial and temporal data. Spatial objects can have arbitrarily complex geometries and can follow complex spatial distributions. Queries over this data are computationally expensive, typically more expensive than common relational queries, making it difficult to perform ad-hoc analyses at interactive speeds. Time series can be arbitrarily long, making it hard to achieve scalable query performance. In addition, the sheer amounts of data that modern applications need to process intensify the above challenges. As a result, existing data management solutions fail to support flexible interactive exploration and are often suitable only for batch-oriented computations. To meet the requirements of spatial and temporal data exploration, new solutions are needed.

1.1 Motivating Applications

The evolution of computing power, combined with advances in data acquisition and the decreasing costs of storage infrastructure, have triggered the emergence of new applications with spatial and temporal data at their core. These modern data-driven applications require rich data exploration capabilities to enable their users to extract useful information and knowledge. In the following, we discuss how leveraging massive-scale spatial and temporal data can (i) significantly advance scientific discovery, and (ii) drastically improve our urban environments. Both of these applications have motivated this thesis.

Scientific discovery. Scientific discovery is no longer solely based on theory and experimentation - the two fundamental paradigms of science. Modern digital sensors and imaging instruments combined with the affordability of high performance computing resources have sparked the creation of additional, new paradigms in science. While theory and experimentation produce mathematical models of the studied phenomena, computational simulations (called the *third paradigm*) allow to further study these models and test how different rules and hypotheses affect their state. Since even simple rules can generate a complex behavior, their understanding relies on analyzing the simulation outputs. As a result, scientists are now overwhelmed by the immense volume of the machine-generated data they need to manage and analyze. The *fourth paradigm* [72] is about leveraging big scientific data generated by simulations, experiments or sensors to push the boundaries of scientific discovery. To realize the fourth paradigm, we need tools for managing and visualizing large amounts of scientific data.

Spatial and temporal data in particular are at the core of data-intensive science as scientists study entities using their morphological or topological properties and examine how these entities evolve over time [70]. In a typical simulation workflow, observational data originating from a variety of input samples is first acquired using a variety of different instruments and techniques, such as bright-field spectroscopy and Magnetic Resonance Imaging (MRI). The massive amounts of collected observational data can then be used in the construction of spatial models of the studied phenomenon. To extract the key properties to be incorporated into the models, scientists examine the data in an exploratory fashion. Neuroscientists in the Human Brain Project (HBP) [106], for example, build structural models that capture different aspects of the brain on different spatial

scales. Once a model is built, it is validated by computing statistical properties for different regions and testing them against the observational data. Simulation tools then leverage the models to foster in-silico experimentation which in turn generates large amounts of spatio-temporal data. Accessing relevant parts of the simulation data quickly is key to scientific discovery; first, potential simulation configuration problems are identified and then the selected data subset is isolated and subjected to more extensive exploratory studies testing different hypotheses.

Urban planning. Urban planning concerns the organization of human activities in residential areas. Specifically, it controls the development and organization of a city, determining the land uses, the rules for building constructions, and the location of public facilities. Urban planning decisions are critical in enabling sustainable and vibrant environments, especially in the face of the rapidly growing urban population. However, urban planning is not simple: cities are complex environments that are shaped by multiple factors [51]. As a result, making urban planning decisions on the basis of experience, precedent and data analyzed in isolation is hard. Nowadays, a growing number of sensors and mobile devices capture the city's momentum and broadcast resident activities, creating new opportunities for data-driven approaches through which the stakeholders can make more informed decisions. At the same time, analyzing ever-increasing volumes of spatio-temporal data from urban environments is challenging.

Visual analytics systems play a key role in the analysis of urban data. They generate comprehensive visual representations (such as heatmaps and parallel coordinate charts [79]) that free up limited cognitive resources allowing to focus on higher-level problems and obtain insights [99]. To support exploratory analyses effectively, visual analytics systems must provide interactive response times, since high latency reduces the rate at which users make observations, draw generalizations and generate hypotheses [99].

With an increasing demand for *effective* exploration of large spatial and temporal data in different domains, data management infrastructure needs to advance. This thesis bridges the gap between data management techniques and the requirements for low latency, flexibility and scalability in exploratory analyses.

1.2 Spatial and Temporal Data Exploration: The Data Management Perspective

Knowledge discovery and decision making are driven by questions. Answering these questions involves data operations that are unknown beforehand and unpredictable. Ad-hoc queries are executed to assess the problem space, unveil data characteristics, verify data quality, and examine different hypotheses [44, 63, 77]. Due to often unclear requirements, designing data management solutions for exploratory use cases is hard. In this thesis, we investigate different exploratory use cases in visual analytics systems and scientific simulation pipelines, identify common types of queries, and focus on designing efficient solutions to support them. The identified exploratory use cases are described next.

1.2.1 Interactive Spatial Data Exploration

As described before, interactivity is key for the visual analysis of spatio-temporal data sets. Among their features, visual analytics systems allow users to visualize data of interest at different resolutions over varying time periods. At the same time, they also enable the visual comparison of several data sets [45].

From a data management perspective, these visualizations are primarily accomplished through spatial aggregation queries that compute an aggregate function over the result of a spatial join between two data sets, typically a set of points and a set of polygons. In the context of a city, the point data may for example correspond to taxi rides, 311 noise complaints, crime, subway or restaurant locations, and the polygonal regions could be neighborhoods or zip codes. To support exploratory analysis, visual analytics systems must be interactive and provide fast response times, which is particularly challenging when we consider the massive volume and complexity of today's spatio-temporal data sets. Spatial aggregation is a fundamental operation in a variety of applications other than visual tools (such as connected mobility applications) that also require near real-time responses.

1.2.2 Ad-hoc Spatial Data Exploration

Data exploration tasks typically involve multiple data sets. The goal of the exploration is to find data sets, and in them data subsets that are most relevant to the analysis. This ad-hoc exploration allows to test a hypothesis on the chosen data subsets, validate statistical properties of the simulated models or visualize certain regions in details [70].

From a data management perspective, the most common way of carrying out such exploratory tasks is to execute a range query on different data sets simultaneously and obtain an answer for each of them, to retrieve all the data related to the region of interest. For example, in the process of comparing the statistical properties of their models against real tissues, neuroscientists use three-dimensional range queries to extract data corresponding to the same brain region from different observational data sources (such as MRI scans of different subjects). With the increase in the number and size of the analyzed data sets the exploration process is significantly hampered.

1.2.3 Scalable Time Series Exploration

Time series data is becoming abundant in a wide range of applications. For instance, many scientific spatial simulations produce time series as output. Typically, not all this data is interesting or important. Ad-hoc exploration allows scientists to identify interesting time series, and in them interesting time intervals, thereby understanding different phenomena that occurred during the simulation. For example, a neuroscientist may only be interested in specific neurons that are firing at specific time steps of the simulation.

This exploration can be performed by issuing two-dimensional range queries (with the two dimensions being value and time) that retrieve the data satisfying a value constraint (e.g., *greater than* a certain constant threshold) for a specified time interval. As both the number and the length of the time series increase, executing such queries on time series becomes more time-consuming.

1.3 The Inadequacy of Current Data Management Approaches

Spatial and temporal data management has been an important research direction for over four decades [54, 67, 102, 119]. However, processing techniques and access methods have been traditionally designed for batch queries [46] and thus fail to meet the requirements of spatial and temporal data exploration. This thesis identifies the technical limitations due to which existing approaches are unsuitable for exploratory applications as follows:

Costly geometric computations. Spatial query operators (such as spatial joins) rely on computationally-intensive, time-consuming geometric tests that severely degrade query performance, especially in regard to the massive volume and complexity of today's data sets. For efficient query processing, we need to minimize (or completely eliminate) these tests, while ensuring the quality of the query results.

Imprecise spatial approximations. The purpose of spatial access methods is to exclude a large subset of spatial objects from the result as early as possible. However, spatial indexes rely on inaccurate filtering predicates (spatial approximations) to describe the data stored in index nodes which has a severe impact on index selectivity. To minimize (or completely avoid) unnecessary accesses to slow mediums (such as HDDs or SSDs) for retrieving objects, we need more effective spatial approximations.

No indexing support for queries over multiple data sets. To support access to multiple data sets, existing spatial techniques create an index either per data set or over the union of all data sets. The first strategy does not scale well with an increasing number of queried data sets, and the second is inefficient when only a small subset of the indexed data sets is queried. To reduce the query cost and decouple it from the total number of indexed data sets, we need index structures that inherently support multiple spatial data sets.

Wide data-to-query gap. Spatial access methods index entire data sets upfront in a time-consuming process before queries can be executed. To enable interactive access to data, we need incremental indexing techniques that minimize upfront costs and adapt to the workload.

No indexing support for queries over multiple data attributes. Time series access methods use single-attribute indexes that index either the time or the value domain and thus cannot effectively prune the search space using constraints on both attributes. For efficient and scalable processing of time series, we need composite indexes that are both time- and value-aware.

1.4 Thesis Statement and Contributions

The goal of this thesis is to bridge the gap between the requirements of spatial and temporal data exploration and the capabilities of existing data management systems. To that end, we focus on query processing techniques and access methods, on which the performance of data management systems depends, and revisit their design to eliminate bottlenecks that oppose the interactivity and scalability requirements of exploratory applications.

Thesis Statement

Modern applications need to explore large amounts of spatial and temporal data at interactive speeds, challenging traditional query processing techniques that rely on time-consuming computations and inefficient access methods. Query operators that exploit specialized hardware and workload-aware access methods enable scalable and interactive exploration of spatial and temporal data.

To address the technical limitations of current approaches and cater for the needs of interactive exploratory applications, this thesis introduces a novel family of spatial and temporal access methods and query operators:

An efficient spatial aggregation operator. We exploit GPU rasterization to avoid costly geometric tests and thereby enable interactive response times for ad-hoc spatial aggregation of a set of data points over a set of arbitrary polygons. As part of the driver provided by the hardware vendors, rasterization is optimized to make use of the underlying architecture and maximize the occupancy of the GPU. Our approach, Raster Join, processes queries on-the-fly, which allows taking into account dynamic updates to query parameters, without requiring any memory-intensive pre-computation. At the same time, it exhibits superior performance compared to index-based approaches.

A bounding object for spatial data. We introduce a novel spatial data approximation method called Clipped Bounding Box (or CBB). We combine a pre-existing and widely used approximation, the Minimum Bounding Box (MBB), with few additional multi-dimensional points that subtract out (clip away) large corner (hyper-)rectangles from the MBB. This improves the approximation quality of the MBB with a low representation and complexity overhead because (i) corners are the convergence points of the dimension-wise maxima/minima of the bounded objects and thus are not fully covered by objects, as the data is distributed differently along the various dimensions, (ii) being the extremities, corners are likely to unnecessarily overlap with queries, and, (iii) clipped corners can be represented compactly. To compute the additional points, we leverage Pareto optimality. We plugged the CBB in different R-tree variants, and achieved a reduction of $\approx 26\%$ in I/Os (across different variants and workloads).

A data structure for multiple spatial data sets. We present STITCH, a novel data structure that indexes multiple spatial data sets that are all enclosed in the same spatial universe. STITCH allows to scalably select spatial regions of interest from a desired subset of indexed data sets, outperforming the state-of-the-art by a factor of up to 12.3.

Incremental data layout organization and indexing of multiple spatial data sets. We present a novel approach (Space Odyssey) that, driven by the actual query needs, minimizes the amount of data that needs to be organized, and optimizes access to data sets frequently queried together. Compared to the state-of-the-art, Space Odyssey accelerates exploratory analysis of spatial data by substantially reducing data-to-query time.

Time- and space-efficient time series indexing. We propose RUBIK, a new indexing scheme for time series data. RUBIK encodes time series values with bitmaps, and compresses the bitmaps using a Quadtree, which allows executing queries with both value and time constraints directly on the compressed representation. Compared to indexing schemes that compress bitmaps using run-length encoding techniques, RUBIK accelerates queries by a factor of between 6 and 23, while producing a more space-efficient index.

Note that the techniques presented in this thesis optimize system behavior in a single-node setting. With multi-core and many-core processors, as well as hardware accelerators such as GPUs, single nodes today have vast computing capabilities. In addition, optimizing single-node behavior is in many cases orthogonal to distributing work to multiple nodes. Therefore our techniques could also provide performance benefits in distributed settings. For instance, many distributed frameworks for spatial data (e.g., Hadoop-GIS [4], SpatialHadoop [47], GeoSpark [171], Simba [169]) rely on MBBs and R-trees for partitioning and indexing. They can thus benefit from our CBBs as well as our workload-aware indexing strategy.

The aforementioned contributions serve as a platform to show the following key intellectual insights:

- We show that GPU rasterization provides a means to evaluate spatial queries efficiently and thus enable interactive data exploration. The rasterization operation converts a geometric primitive (i.e., a polygon) into a fine-grained approximation in terms of pixels on-the-fly, which allows to avoid expensive geometric computations in most cases.
- Minimum Bounding Boxes (MBBs) are compact to store and cheap to query, but approximate poorly today's complex real-world data. In contrast, other spatial approximations proposed in prior work (such as the convex hull [26]) approximate data tightly, but are not time- and space- efficient. We show that by subtracting out (clipping away) the corners of MBBs, we obtain a tight yet simple spatial approximation.
- Workload-aware data layout organization and indexing are key in enabling exploratory access to (multiple) spatial data sets. Organizing spatial objects not only based on their spatial proximity, but also based on the data category (i.e., data set) that they belong to,

facilitates efficient access to categories and regions of interest. Furthermore, building (partial) data structures incrementally only for the bits of the data needed decreases data-to-query time.

- We argue that preserving spatial and temporal proximity in time series access methods results in improving both space-efficiency and query performance. This is because (i) time series (obtained through sensors, or produced by simulations) often exhibit similarity in both space and time (i.e., values of consecutive time steps can be similar and time series in the same spatial vicinity can have similar value patterns) and (ii) data access patterns frequently align with temporal or spatial proximity.

1.5 Thesis Outline

This section outlines the thesis and summarizes the content of each chapter.

Chapter 2 presents general background information on the topics of this thesis. Each of the subsequent chapters presents a solution for a data management problem related to the exploration of spatial or temporal data. The addressed problem is introduced in detail in each chapter, along with additional necessary background information and related work.

In Chapter 3 we describe Raster Join [149], a GPU-based approach that provides interactive response times to computationally-intensive spatial aggregation queries over arbitrarily-shaped regions, which is an essential operation in visual analytics systems. State-of-the-art techniques for interactive exploration rely on pre-aggregation and are thus unsuitable in this setting because they fix the query constraints and do not provide support for polygons of arbitrary shape. To address this limitation, Raster Join converts a spatial aggregation query into a set of drawing operations on a canvas. Interactive speeds are achieved by leveraging the rendering pipeline of the graphics hardware (GPU), and in particular the *rasterization* operation, which converts a polygon into a collection of pixels and is natively supported by GPUs. Raster Join evaluates queries on-the-fly, allows to trade-off accuracy for response time by adjusting the canvas resolution, and can even provide accurate results when combined with a polygon index. In this work, we make a case for incorporating computer graphics techniques in spatial databases, based on the observation that both domains address analogous problems.

In Chapter 4 we discuss challenges and performance bottlenecks associated with processing spatially extended three-dimensional objects. The majority of spatial data processing techniques use Minimum Bounding Boxes (MBBs) to represent a collection of spatial objects. The MBB is the smallest axis-aligned box that encloses a given set of spatial objects. Access methods of the R-tree family, for example, employ a hierarchy of MBBs to recursively enclose the indexed spatial objects. Minimum Bounding Boxes have the benefits of being computation-efficient (they are computed simply with linear cost, while overlap/intersection tests are cheap) and space-efficient. However, fitting (groups of) spatial objects into a rough box often results in a very poor approximation of the underlying data. The resulting MBBs contain a lot of

"dead space"—fragments of bounded area that contain no actual objects—that can significantly reduce the filtering efficacy. To address this disadvantage of MBBs, we introduce the concept of *Clipped* Bounding Boxes (CBBs) [135]. Essentially, a CBB augments a MBB by "clipping" away dead space that is concentrated around the corners of the box. By doing so, the quality of the approximation is improved, and unnecessary recursions in dead space are avoided, which results in improved query performance.

Chapter 5 discusses the challenge of efficiently querying an arbitrary combination of spatial data sets, while returning the query result on each data set. Indexing and querying the data sets separately is only efficient when the number of queried data sets is small, while using a single index for all data sets incurs considerable overhead when only a small subset needs to be retrieved. To address this challenge, we introduce STITCH, an index structure for the scalable execution of spatial range queries on multiple data sets. The leaf level of the structure is data set-aware: within a leaf node, all the objects belonging to the same data set are groupped together. On the other side, the internal levels are data set-oblivious; they only index the spatial universe that encloses all different data sets. In addition, this chapter presents Space Odyssey [125], an incremental indexing approach that continuously optimizes access to data sets frequently queried together and thereby enables ad-hoc exploration.

To facilitate the exploration of time series data, we provide support for time-bound threshold queries. A time-bound threshold query returns all the values that occurred within specified time bounds and are above a given threshold. Range-encoded bitmap indexes are state-of-the-art when it comes to querying read-only data. However, the run-length encoding-based compression schemes that are commonly applied to the bitmaps severely impact the performance of time-bound threshold queries, as they disrupt the one-to-one mapping between time steps and bit positions. In Chapter 6 we propose RUBIK [150], a technique that encodes time series values as bitmaps and groups similar bitmaps together to exploit similarities between time series. It then applies a quadtree-based compression scheme to each group, which leverages similarities within the time series. As a result, RUBIK enables searching compressed bitmaps along both the value and the time dimensions.

Chapter 7 summarizes the thesis and discusses future research directions.

2 Background

In this chapter, we introduce key terms and concepts in spatial and temporal data management. We describe spatial queries involved in exploration tasks, the general design of spatial query processing strategies, spatial access methods used to scale down the result search space, and time series access methods used to efficiently retrieve time series data.

More detailed discussions of the state-of-the-art concerning specific problems addressed in this thesis can be found in the respective chapters.

2.1 Spatial Data Representation

Spatial features are represented as either *rasters* or *vectors*. The raster format represents the study area with a n-dimensional regular grid. The location of the features in the study area is depicted by the values (e.g., intensity or color) in the cells overlaying the features. Vector data types on the other hand, represent spatial features with their geometry. For example, geographic features are represented using points, lines and polygons, while structures in scientific simulations can be volumetric cones, and triangular or tetrahedral meshes. This thesis focuses on processing vector data types, mainly in 2D and 3D. Therefore, the term spatial data is used to refer to data tuples that contain a geometric component in a n-dimensional Euclidean space or a suitable subspace thereof.

2.2 Temporal Data Representation

Temporal data denotes the evolution of an object characteristic over time. The time component in temporal data can be either implicit or explicit. If the time is explicit, then each value is associated with a time stamp. If the time is implicit, the actual time stamp is not important. What is important is the ordering of the values, i.e., how the values are connected in a sequence. This thesis focuses on a specific kind of temporal data, time series. *Time series* are sequences of

values generated (or collected) as time progresses and are frequently encountered in different applications, from finance (e.g., stock prices) and meteorology to simulation sciences.

2.3 Processing Spatial Queries

The most common query types used in spatial data exploration are range queries, k-nearest neighbor queries, and spatial joins.

Range Query. Range queries are typically defined by a $n-$dimensional hyperrectangle. Given this hyperrectangle R, a set of spatial objects M, and a spatial predicate θ, a range query retrieves all the objects of M that satisfy the predicate θ with respect to R. Typical spatial predicates are intersection (getting all objects overlapping the hypervolume of R, fully or partially), containment (objects strictly contained within the bounds of R), and enclosure (objects strictly enclosing R). In scientific simulations, range queries are used to retrieve parts of the model data in particular regions of interest.

k-Nearest Neighbor Query. Given a set of spatial objects M and a particular object o, a k-nearest neighbor query returns (at most) k objects from M that are closest to object o.

Spatial Join Query. Given two sets of spatial objects M_1 and M_2 and a spatial predicate θ, a spatial join query yields all pairs of objects ($o_1 \in M_1, o_2 \in M_2$) that satisfy θ. The most frequently used spatial predicate θ is intersection, but containment or enclosure can also be applied. Spatial joins are essential in many applications. For example, connected mobility companies such as Uber need to join locations of passenger requests with a set of polygonal regions to display available cars and enable dynamic pricing [153].

Processing spatial queries is challenging, mainly for two reasons. First, there is no total order among spatial objects that preserves spatial proximity, i.e., it is impossible to totally order spatial objects linearly, so that spatially close objects in the multidimensional domain, are also close in the linear order. Consequently, algorithms designed for relational (one-dimensional) data cannot be directly applied to handle spatial data. Second, spatial objects have an arbitrarily complex structure. Even worse, different spatial objects can have very different structures (e.g., a point is different from a polygon).

To address the lack of total ordering, specialized multidimensional access methods have been proposed (see Section 2.4). To address the second challenge, spatial query processing is predominantly based on a two-step "*filter and refine*" approach [54]. The query is first solved using approximations of the geometries, and then false matches are removed by examining the actual geometries. The approximation usually serves as a general representation for different geometries, which makes the filtering step geometry-oblivious. In addition, the approximation has a significantly simpler structure than the actual object, which reduces computation and storage costs.

The most widely used spatial object approximation is the *Minimum Bounding Rectangle* (MBR - in 2D) or *Minimum Bounding Box* (MBB - in 3D). The MBR is the smallest axis-aligned rectangle that encloses the complete geometry of an object and can represent objects (or sets of objects) with different geometries (from points to complex volumetric shapes).

The refinement phase performs exact geometric tests. The performed tests depend on the object geometries and the spatial predicate. Since real-world data often consists of points (e.g., GPS locations) and polygons (e.g., geographical region boundaries), a commonly applied geometric test is the *point-in-polygon test* that determines whether a given point lies inside a polygon. The most widely used point-in-polygon test algorithm [116] (called ray casting, also known as the crossing number algorithm or the even–odd rule algorithm) shoots a ray from the point, and counts the number of proper intersections of the ray with the polygon edges. If the count is odd, the point is inside the polygon. The complexity of this algorithm is linear with the size of the polygon (number of vertices).

2.4 Spatial Access Methods (SAMS)

Spatial query processing relies on spatial access methods to achieve efficiency. Traditional techniques designed for one-dimensional data could be applied to (extended) spatial objects by first transforming the objects to points in some parameter space. However, this transformation does not preserve the spatial neighborhood of objects in the original space, and thus the distribution of parameter points can be extremely skewed, which harms query performance. Therefore, access methods have been designed specifically for spatial data.

Spatial access methods are typically built on top of object approximations (e.g., MBRs) and not the objects themselves. Consequently, they act as a filter, creating a candidate set that may include false matches. Nevertheless, they still allow to filter out a large number of objects as in general a small and locally restricted part of the space is relevant to the query.

Spatial indexes can be categorized in two classes according to the partitioning strategy that they employ: *space-oriented* [6, 18, 80, 129, 155], and *data-oriented* [12, 15, 17, 66, 103, 131]. Space-oriented indexes partition the entire space enclosing the data into disjoint (i.e., non-overlapping) regions. The objects are associated with each partition they intersect and thus an object can be stored in many partitions. As a result of this multiple assignment strategy, query performance may degrade significantly. This is because: 1) the same object might be retrieved and processed multiple times, which increases the amount of data transferred and the computational requirements, 2) some results may be detected more than once, and therefore an additional post-processing step is needed for removing redundant results, 3) configuring the partitioning resolution is challenging, because increasing the number of partitions also increases the replication rate. In contrast, data-oriented indexes assign each object to exactly one region and allow regions to overlap. This means that there may exist several regions potentially containing the searched object.

2.5 Time Series Access Methods

Time series can be queried to retrieve, for example, specific sub-sequences, extrema points, or points in time at which the time series had a specific value. To make these queries efficient, we need index structures.

A time series can be seen as a point in a multi-dimensional space, where the number of dimensions corresponds to the length of the time series. Consequently, a time series can be indexed using a Spatial Access Method (SAM). However, approximating time series data with MBRs in SAMs, does not capture well their sequential nature. The resulting MBRs are very wide, and have a lot of overlap with empty space. This causes many queries to unnecessarily overlap with them. In addition, typical time series are very long, which makes Spatial Access Methods suffer from the curse of dimensionality. This problem is addressed by first performing a dimensionality reduction on the time series before indexing them with a SAM. Even better, we can use indexes specifically designed for higher-dimensional spaces (such as the X-tree [19]) to index the reduced time series representation. Apart from techniques that rely on SAMs to provide efficient indexing, specialized time series indexes have also been proposed [13, 30, 31].

Techniques that rely on dimensionality reduction cannot be applied for queries accessing specific values of the time series that satisfy a user-defined condition, such as range queries that retrieve time segments at any series with values above a given threshold. This is because in a reduced feature space, the original intervals where the time series values are above a given threshold cannot be generated.

To support ah-hoc queries accessing specific time series values, we can use a *bitmap index*. Bitmap indexes are particularly useful for data that is read-only [34], which is the case in many applications of time series. In scientific simulations, for example, once the simulation is over, the generated results are not further updated. Bitmap indexes organize the data as a two-dimensional table of 0's and 1's. Each row of the table represents a data record, and each column represents a value or a range of values. At the intersection of rows and columns there is a single bit that indicates whether the record of the corresponding row has the value (or is within the range of values) denoted in the column. To be effective on large data, this table has to be compressed. With some recent exceptions [94], bitmap compression algorithms typically employ variants of *run-length encoding* [10, 168], i.e., they replace repeated runs of 0's or 1's in the columns of the bitmap index by a single instance of the symbol and a run count.

3 GPU Rasterization for Interactive Spatial Data Exploration

This chapter[1] shows that interactive visual exploration of spatial data relies heavily on spatial aggregation queries that slice and summarize the data over different regions. These queries require computationally-intensive point-in-polygon tests that associate data points to polygonal regions, challenging the responsiveness of visualization tools. This challenge is compounded by the sheer amounts of data, requiring a large number of such tests to be performed. Traditional pre-aggregation approaches are unsuitable in this setting since they fix the query constraints and support only rectangular regions. Contrarily, query constraints are defined interactively in visual analytics systems, and polygons can be of arbitrary shapes.

In this chapter, we convert a spatial aggregation query into a set of drawing operations on a canvas and leverage the rendering pipeline of the graphics hardware (GPU) to enable interactive response times. Our technique trades accuracy for response time by adjusting the canvas resolution, and can provide accurate results when combined with a polygon index. We evaluate our technique on two large real-world data sets, exhibiting superior performance compared to index-based approaches.

3.1 Introduction

The explosion in the number and size of spatio-temporal data sets from urban environments (e.g., [35, 113, 145]) and social sensors (e.g., [115, 148]) creates new challenges for analyzing this data. The complexity and cost of evaluating queries over space and time for large volumes of data often limits analyses to well-defined questions, what Tukey described as *confirmatory data analysis* [147], typically accomplished through a batch-oriented pipeline. To support *exploratory analyses*, systems must provide interactive response times, since high latency reduces the rate at which users make observations, draw generalizations and generate hypotheses [99].

[1]The material of this chapter has been the basis for the PVLDB 2017 paper *GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons* [149] and the SIGMOD 2018 demo paper *Interactive Visual Exploration of Spatio-Temporal Urban Data Sets using Urbane* [45].

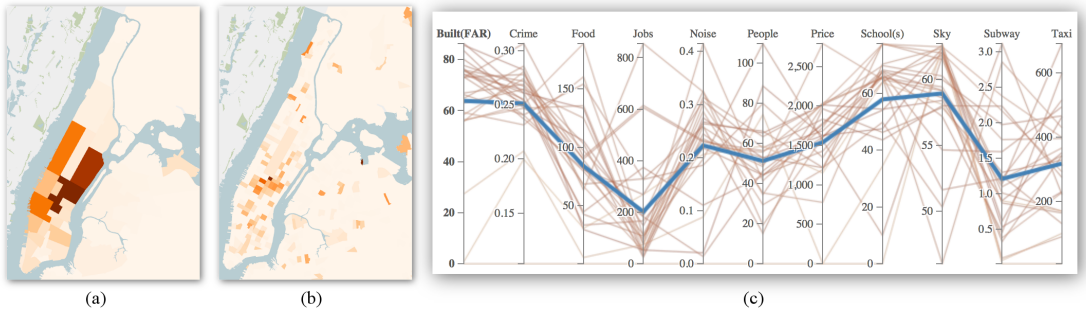


Figure 3.1 – Exploring urban data sets using Urbane: (a) visualizing data distribution per neighborhood, (b) visualizing data distribution per census tract, (c) comparing data over different neighborhoods. The blue line denotes the NYC average for these data sets.

Not surprisingly, the problem of providing efficient support for visualization tools and interactive queries over large data has attracted substantial attention recently, predominantly for relational data [1, 14, 78, 83, 84, 98, 100, 139, 165]. While methods have also been proposed for speeding up selection queries over spatio-temporal data [46, 173], these do not support interactive rates for aggregate queries, that slice and summarize the data in different ways, as required by visual analytics systems [9, 51, 117, 130, 141, 166].

Motivating Application: Visual Exploration of Urban Data Sets. In an effort to enable urban planners and architects to make data-driven decisions, Ferreira et al. developed Urbane, a visualization framework for the exploration of several urban data sets [51]. The framework allows the user to visualize a data set of interest at different resolutions and also enables the visual comparison of several data sets.

Figures 3.1(a) and 3.1(b) show the distribution of NYC taxi pickups (*data set*) in the month of June 2012 using a heat map over two resolutions: neighborhoods and census tracts. To build these heatmaps, aggregate queries are issued that count the number of pickups in each neighborhood and census tract. Through its visual interface, Urbane allows the user to change different parameters dynamically, including the time period, the distribution of interest (e.g., count of taxi pickups, average trip distance, etc.), and even the polygonal regions. Figure 3.1(c) shows multiple data sets being compared using a single visualization: a parallel coordinate chart [79]. In this chart, each data set (or dimension) is represented as a vertical axis, and each region (neighborhood) is mapped to a polyline that traverses across all of the axes, crossing each axis at a position proportional to its value for that dimension. Note that each point in an axis corresponds to a different aggregation for the selected time range for each neighborhood, e.g., Taxi reflects the number of pickups, while Price shows the average price of a square foot. This visual representation is effective for analyzing multivariate data, and can provide insights into the relationships between different indicators. For example, by filtering and varying crime rates, users can observe related patterns in property prices and noise levels over the different neighborhoods.

Motivating Application: Interactive Urban Planning. Policy makers frequently rezone different parts of the city, not only adjusting the zonal boundaries, but also changing the various laws (e.g., new construction rules, building policies for different building types). During this process, they are interested in viewing how the other aspects of the city (represented by urban data sets) vary with the new zoning. This operation typically consists of users changing polygonal boundaries, and inspecting the summary aggregation of the data sets until they are satisfied with a particular configuration.

In this process, urban planners may also place new resources (e.g., bus stops, police stations), and again inspect the coverage with respect to different urban data sets. The coverage is commonly computed by using a restricted Voronoi diagram [20] to associate each resource with a polygonal region, and then aggregating the urban data over these polygons. To be effective, these summarizations must be executed in real-time as configurations change.

Problem Statement and Challenges. In this chapter, we propose new approaches to speedup the execution of spatial aggregation queries, which, as illustrated in the examples above, are essential to explore and visualize spatio-temporal data. These queries can be translated into the following SQL-like query that computes an aggregate function over the result of a spatial join between two data sets, typically a set of points and a set of polygons.

```
SELECT AGG(a_i) FROM P, R
WHERE P.loc INSIDE R.geometry [AND filterCondition]*
GROUP BY R.id
```

Given a set of points of the form $P(loc, a_1, a_2, \dots)$, where loc and a_i are the location and attributes of the point, and a set of regions $R(id, geometry)$, this query performs an aggregation (`AGG`) over the result of the join between P and R. Functions commonly used for `AGG` include the count of points and average of the specified attribute a_i. The `geometry` of a region can be any *arbitrary polygon*. The query can also have zero or more `filterConditions` on the attributes. In general, P and R can be either tables (representing data sets) or the results from a sub-query (or nested query).

The heat maps in the Figures 3.1(a) and 3.1(b) were generated by setting P as pickup locations of the taxi data; R as either neighborhood (a) or census tract (b) polygons; `AGG` as COUNT(*); and filtered on time (June 2012). To obtain the parallel coordinate visualization in Figure 3.1(c), multiple queries are required: *the above query has to be executed for each of the data sets of interest that contribute to the dimensions of the chart.*

Enabling fast response times to such queries is challenging for several reasons. First, the point-in-polygon (PIP) tests to find which polygons contain each point require time linear with respect to the size of the polygons. Real-world polygonal regions have complex shapes, often consisting of hundreds of vertices. This problem is compounded due to the fact that data sets can have hundreds of millions to several billion points. Second, as illustrated in the examples above, when using interactive visual analytics tools, users can dynamically change not only the filtering conditions

and aggregation operations, but also the polygonal regions used in the query. Since *the query rate is very high* in these tools, delays in processing a query have a snowballing effect over the response times.

Existing spatial join techniques, common in database systems, are costly and often suitable only for batch-oriented computations. The join is first solved using approximations (e.g., bounding boxes) of the geometries. Then, false matches are removed by comparing the geometries (e.g., performing PIP tests), which is a computationally expensive task. This two stage evaluation strategy also introduces the overhead of materializing the results of the first stage. Finally, the aggregates are computed over the materialized join results and incur additional query processing costs. Data cube-based structures (e.g., [98]) can be used to maintain aggregate values. However, creating such structures requires costly pre-processing while the memory overhead can be prohibitively high. More importantly, these techniques *do not support queries over arbitrary polygonal regions*, and thus are unsuitable for our purposes.

Last but not least, while powerful servers might be accessible to some, many users have no alternative other than commodity hardware (e.g., business grade laptops, desktops). Having approaches to efficiently evaluate the above queries on commodity systems can help democratize large-scale visual analytics and make these techniques available to a wider community.

For visual analytics systems, approximate answers to queries are often sufficient as long as they do not alter the resulting visualizations. Moreover, the exploration is typically performed using the “level-of-detail" (LOD) paradigm: first look at the overview, and then zoom into the regions of interest for more details [133]. Thus, these systems can greatly benefit from an approach that trades accuracy for response times, and enables LOD exploration that improves accuracy when focusing on details.

Our Approach. By leveraging the massive parallelism provided by current generation graphics hardware (Graphics Processing Units or GPUs), we aim to support interactive response times for spatial aggregation over large data. However, accomplishing this is challenging. Since the memory capacity of a GPU is limited, data must be transferred between the CPU and GPU, and this introduces significant overhead when dealing with large data. In addition, to best utilize the available parallelism, GPU occupancy must be maximized. We propose rasterization-based methods that use the following key insights to overcome the above challenges:

- *Insight 1:* It is not necessary to explicitly materialize the result of the spatial join since the final output of the query is simply the aggregate value;
- *Insight 2:* A spatial join between two data sets can be considered as “drawing" the two data sets on the same canvas, and then examining their intersections; and
- *Insight 3:* When working with visualizations, small errors can be tolerated if they cannot be perceived by the user in the visual representation.

Insight 1 allows combining the aggregation operation with the actual join. The advantages of this are twofold: (i) no memory needs to be allocated for storing join results, allowing the GPU

to process more input data, and thus computing the result in fewer passes; and (ii) since no materialization (and corresponding data transfer overhead) is required, query times are improved. Insight 2 allows us to frame the problem of evaluating spatial aggregation as renderings, using operations that are highly optimized for the GPU. In particular, it allows us to exploit the *rasterization* operation, which converts a polygon into a collection of pixels. Rasterization is an important component of the graphics rendering pipeline and is natively supported by GPUs. As part of the driver provided by the hardware vendors, rasterization is optimized to make use of the underlying architecture and thus maximize occupancy of the GPU. By allowing approximate results, Insight 3 enables a mechanism to completely avoid the costly point-in-polygon tests, and use *only* the drawing operations, thus leading to a significant performance improvement over traditional techniques. Moreover, it allows an algorithmic design in which the input data is transferred *only once* to the GPU, further reducing the memory transfer overhead.

Even though our focus in this work is to enable seamless interaction on visual analysis tools, we would like to note that the spatial aggregation has utility in a variety of applications in several fields. For example, this type of query is commonly used to generate scalar functions for topological data analysis [36, 43, 108]. While these applications might not require interactivity per se, having fast response times would definitely improve analysis efficiency.

Contributions. Our contributions can be summarized as follows:

- Based on the observation that spatial databases rely on the same primitives (e.g., points, polygons) and operations (e.g., intersections) common in computer graphics rendering, we *develop spatial query operators that exploit GPUs and take advantage of their native support for rendering*.
- We *propose bounded raster join, an efficient approximate approach, that by eliminating the need for costly point-in-polygon tests provides close to accurate results in real-time*.
- We *develop an accurate variant of the bounded raster join that combines an index-based join with rasterization to efficiently evaluate spatial aggregation queries*.

To the best of our knowledge, this is the first work that efficiently evaluates spatial aggregation using rendering operations. The advantages of blending computer graphics techniques with database queries are amply clear from our comprehensive experimental evaluation using two real world data sets– NYC taxi data (~868 million points) and geo-tagged Twitter (~2.2 billion points). The results show that the bounded raster join obtains over two orders of magnitude speedup compared to an optimized CPU approach when the data fits in main memory (note that the data need not fit in GPU memory), and over 30X speedup otherwise. In fact, it can execute queries involving over 868 million points in only 1.1 second even on a current generation laptop. We also report the accuracy-efficiency as well as bound-error trade-offs of the bounded approach, and show that the errors incurred using even a very coarse bound do not impact the quality of the generated visual representations. This makes our approach extremely valuable for visualization-based exploratory analysis where interactivity is essential. Given the widespread availability of GPUs on desktops and laptops, our approach brings large-scale analytics to commodity hardware.

3.2 Related Work

Spatial Aggregation. To support interactive response times for analytical queries in visualization systems, compact data structures such as Nanocubes [98] and Hashedcubes [118] have been designed to store and query the CUBE operator for spatio-temporal data sets. These techniques mainly rely on static pre-computation: they pre-aggregate records at various spatial resolutions and store this summarized information in a hierarchy of rectangular regions (maintained using a quadtree). To enable filtering and aggregation support over different attributes, these attributes must be known at build-time to be included as a dimension of the cube. Also, the granularity of the filtering depends on the number of discrete ranges the attribute is divided into. Thus, supporting filtering and aggregation over arbitrary attributes not only entails substantial pre-computation costs, but also *exponentially* increases the storage requirements, often making it impractical for real-world, large data sets. More importantly, since these structures maintain aggregate information over a hierarchy of rectangular regions, they have three key limitations: 1) the queries supported are constrained to only rectangular regions; 2) spatial aggregation has to be executed as a collection of queries, one for each region, which is inefficient for a large number of regions; and 3) the computed aggregates are approximate and the error cannot be dynamically bounded (since the accuracy depends on the quadtree resolution). Supporting arbitrary polygons and obtaining accurate results requires accessing the raw data (which might require additional spatial indexes) and defeats the purpose of maintaining a cube structure.

Several algorithms have also been proposed by the database community to evaluate spatial aggregate queries [143, 158]. For instance, the aRtree [121] enhances the R-tree [66] structure by keeping aggregate information in intermediate nodes. These algorithms rely on annotated data structures and thus suffer from the aforementioned key limitations. Besides, they only support a spatial range selection predicate and do not support predicates on other attributes which makes them unsuitable for a dynamic setting. Closest to our approach are online aggregation techniques for spatial databases. However, prior work in the area [160] is also limited to range queries and does not provide support for join and group-by predicates.

Spatial Joins on CPUs. Our work is closely related to spatial join techniques since the join operation is the most expensive component of spatial aggregation queries. However, recall that explicit materialization of the join results is not required. Spatial joins typically involve two steps: filtering followed by refinement. The filtering step finds pairs of spatial elements whose approximations (minimum bounding rectangles - MBRs) intersect with each other, while the refinement step detects the intersection between the actual geometries. Past research on spatial join algorithms has largely focused on the filtering step [28, 81, 122, 123]. To improve the processing of spatial queries over complex geometries, the Rasterization Filter [176] approximates polygons with rectangular tiles and serves as an additional filtering step that reduces the number of costly geometry-geometry comparisons. This approximation is calculated statically and stored in the database. In contrast, our approach exploits GPU rasterization to produce a fine-grained polygonal approximation on-the-fly and completely eliminates MBR-based tests. Apart from the aforementioned standalone solutions, several commercial and freely available DBMSs offer

spatial extensions complying with the two-step evaluation process [40, 42, 50, 110, 127, 159]. While the filtering step is usually efficient, the refinement often degrades query performance since it involves costly computational geometry algorithms [137]. As a point of comparison, we performed a join between only 10 NYC neighborhood polygons and the taxi data using a commercial database. The query took over ten minutes to execute. This performance is not suitable for interactive visual analytics systems. More recently, distributed solutions such as Hadoop-GIS [4] and Simba [169] were proposed for spatial query processing. Both these solutions suffer from network bottlenecks which might affect interactivity, and also rely on the presence of powerful clusters for processing. Hadoop-based solutions are further constrained due to disk I/O. As we show in our experiments, our approach attains interactive speeds using GPUs that are ubiquitous in current generation desktops and laptops.

Spatial Query Processing on GPUs. Over the past decade, several research efforts have leveraged programmable GPUs to boost the performance of general, data-intensive operations [11, 49, 58, 69, 90]. Earlier techniques (e.g., [58]) employed the programmable rendering pipeline to execute these queries. Due to a fixed set of operations supported by the pipeline, it often resulted in overly complex implementations to work around the restrictions. With the advent of more flexible GPGPU interfaces, there have been several full fledged GPU-accelerated RDBMSs [24, 104]. MapD [104] accelerates SQL queries by compiling them to native GPU code and leveraging GPU parallelism. It is a relational database that currently does not support polygonal queries[2]. On the other hand, exploiting graphics processors for spatial databases is natural, as it involves primitive types (geometric objects) and operations (spatial selections, containment tests) that are similar to the ones used in graphics. However, there has been limited amount of work in this area. Sun et al. [140] used GPU rasterization to improve the efficiency of spatial selections and joins. In the case of joins, they used rasterization as part of the join refinement phase to determine if two polygons do not intersect. However, this approach does not scale with an increasing number of polygons since the GPU is only used to perform pairwise comparisons. In contrast, the rasterization pipeline plays an integral part in our technique. By exploiting the capabilities of modern GPUs, we are able to perform more complex operations at faster speeds.

Closest to our work, Zhang et al. [172, 174] used GPUs to join points with polygons. They index the points with a Quadtree to achieve load balancing and enable batch processing. In the filtering step of the join, the polygons are approximated using MBRs. Zhang et al. [173] used the spatial join technique proposed in [172] to pre-compute spatial aggregations in a predefined hierarchy of spatial regions on the GPU. In contrast, we perform the aggregation on-the-fly, taking into account dynamic constraints. More recently, they extended their spatial join framework [175] to handle larger point data sets. As they materialize the join result, to optimize the memory usage, they make the limiting assumption that no two polygons intersect, thus ensuring the join size is at most the size of the input. Additionally, to improve efficiency, they truncate coordinates to 16-bit integers, thus resulting in approximate joins as well. Because we focus on analytical

[2]MapD currently has only one GIS function: https://www.mapd.com/docs/latest/mapd-core-guide/dml/

queries that do not require explicit materialization of the join result, we can use rasterization to better approximate the polygons as well as combine the join with the aggregation operation.

Aghajarian et al. [3] employ the GPU to join non-indexed polygonal data sets. The focus of our work, however, is aggregating points contained within polygonal regions. Doraiswamy et al. [46] proposed a customized kd-tree structure for GPUs that supports arbitrary polygonal queries. While the proposed index provides interactive response times for selection queries, the evaluation of the join requires one selection to be performed for each polygon, and is thus inefficient when the polygon data set is large.

3.3 Background: Graphics Pipeline

The most common operation in graphics-intensive applications (e.g., games) is to render a collection of triangular and polygonal meshes that make up a scene. To achieve *interactivity*, such applications rely heavily on rasterization and approximate visual effects (e.g., shadows) to render the scenes. Modern GPUs exhibit impressive computational power (the latest Nvidia GTX 1080 Ti reaches 10.6 TFLOPS) and implement rasterization in hardware to speedup the rendering process. The key idea in our approach is to leverage the graphics hardware rendering pipeline for rasterization and the efficient execution of spatial aggregation queries.

Rasterization-based Graphics Pipeline. Rendering a collection of triangles is accomplished in a series of processing stages that compose a graphics pipeline. First, the coordinates of all the vertices (of the triangles) that compose the scene are transformed into a common world coordinate system, and then projected onto the screen space. Next, triangles falling outside the screen (also called *viewport*) are discarded, while those partially outside are *clipped*. Parts of triangles within the viewport are then *rasterized*. *Rasterization* converts each triangle in the screen space into a collection of *fragments*. Here, a *fragment* can be considered as the data corresponding to a pixel. The fragment size therefore depends on the *resolution* (the number of pixels in the screen space). For example, a 800×600 rendering of a scene has fewer pixels (480k pixels) than a high resolution rendering (e.g., $1920 \times 1080 \approx 2$M pixels), and thus has a bigger fragment size. In the final step, each fragment is appropriately colored and displayed onto the screen. Figure 3.2 shows an example where a triangle is rasterized (pixels colored violet) at a given resolution.

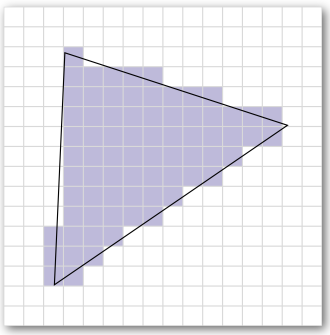


Figure 3.2 – Rasterizing a triangle into pixels.

OpenGL [134], a cross platform rendering API, supports the ability to program parts of the rendering pipeline with a shading language (GLSL), thus allowing for custom functionality. In particular, the custom rendering pipeline, known as *shader programs*, commonly consists of a *vertex shader* and a

fragment shader. The vertex shader allows modifying the first stage of the pipeline, namely, the transformation of the set of vertices to the screen space. The clipping and rasterization stages are handled by the GPU (driver). Finally, the fragment shader allows defining custom processing for each fragment that is generated. Both shaders are executed using a single program, multiple data (SPMD) paradigm.

Rasterization. Given the crucial part it plays in the graphics pipeline, parallel rasterization has had a long research history, as can be seen from these classical papers [109, 126]. Hardware vendors optimize parallel rasterization by directly mapping computational concepts to the internal layout of the GPU. While the details of the rasterization approaches used in current GPU hardware are beyond the scope of this work, we briefly describe the key ideas.

Hardware drivers typically use a variation of the algorithm proposed by Olano and Greer [114]. As a key optimization, they focus on the rasterization of triangles instead of general polygons. The triangle is the simplest convex polygon, and it is thus computationally efficient to test whether a pixel intersects with it. The intersection tests are typically performed by solving linear equations, called edge functions [114]. The rasterization algorithm allows to test whether pixels lie within a given triangle in parallel, and thus is amenable to hardware implementation.

Triangulation. Rendering polygons on the GPU is often accomplished by decomposing them into a set of triangles, an operation called *triangulation*. The problem of polygon triangulation has a rich history in the computational geometry domain. The two most common approaches for triangulation are the ear-clipping algorithm [20] and Delaunay triangulation, in particular, a constrained Delaunay triangulation [132]. Delaunay-based approaches have the advantage of providing theoretical guarantees regarding the quality of the generated triangles (such as minimum angle), and are often preferred for generating better triangle meshes. In this work, we employ constrained Delaunay polygon triangulation.

Frame buffer objects (FBO). Instead of directly displaying the rendered scene onto a physical screen (monitor), OpenGL also allows outputting the result into a “virtual" screen. The virtual screen is represented by a *frame buffer object* (FBO) having a resolution defined by the user. Even though the resolutions supported by existing monitors are limited, current graphics generation hardware supports resolutions as large as 32K $\times$ 32K. Each pixel of the FBO is represented by 4 32-bit values, $[r, g, b, a]$, corresponding to the red, blue, green, and alpha color channels. Users can also modify the FBO to store other quantities such as depth values. Since our goal is to compute the result of a spatial aggregation, we do not make use of any physical screen, but we make extensive use of FBOs to store intermediate results.

3.4 Raster Join

Existing techniques execute spatial aggregation for a given set of points and polygons in two steps: (1) the spatial join is computed between the two data sets; and (2) the join results are aggregated.

Such an approach has two shortcomings. The join operation is expensive, in particular, the PIP tests it requires – in the best-case scenario, one PIP test must be performed for every point-polygon pair that satisfies the join condition, and the complexity of each PIP test is linear with the size of the polygon. Even when the PIP tests are executed in parallel on the GPU, queries still require several seconds to execute even for a relatively small number of points (see Section 3.7 for details). To compute the aggregate as a second step, the join must be materialized. Consequently, given the limited memory on a GPU, the join has to be performed in batches, which incurs additional memory transfer between the CPU and GPU.

In this section, we first discuss how the rasterization operation can be applied to overcome the above shortcomings. We then propose two algorithms: bounded and accurate raster join, which produce approximate and exact results, respectively.

3.4.1 Core Approach

The design of raster join builds on two key observations:

1. A spatial join between a polygon and a point is essentially the intersection observed when the polygon and point are *drawn* on the same canvas.
2. Given that the goal of the query is to compute aggregates, if the join and aggregate operations are combined, there is no need to materialize the join results.

Intuitively, our approach *draws* the points on a canvas and keeps track of the intersections by maintaining *partial aggregates* in the canvas cells. It then *draws* the polygons on the same canvas, and computes the aggregate result from the partial aggregates of the cells that intersect with each polygon. The above operations are accomplished in two steps as described next. To illustrate our approach, we use the following example. We apply the query:

```
SELECT COUNT(*) FROM D_pt, D_poly
WHERE D_poly.region CONTAINS D_pt.location
GROUP BY D_poly.id
```

to the data sets shown in Figure 3.3, D_{poly} with 3 polygons, and D_{pt} with 33 points.

Step I. Draw points: The first step renders the points onto an FBO as shown in Procedure 1. We maintain an array A of size equal to the number of polygons, which is 3 for the example in Figure 3.3. This array is initially set to 0. When a point is processed, it is first transformed into the screen space, and then rasterization converts it into a fragment that is rendered onto an FBO. In this FBO, we use the color channels of a pixel for storing the count of points falling in that pixel. Instead of setting a color to that pixel, we *add* to the color of the pixel (e.g., the red channel of the pixel is incremented by 1). OpenGL only allows specifying colors for a fragment in the fragment shader. The way the specified color is combined with that in the FBO is controlled by a *blend function*. We set this function such that the specified color is added to the existing color

in the FBO. This step results in F_{pt}, an FBO storing the count of points that fall into each of its pixels. The FBO for the input in Figure 3.3 is illustrated in Figure 3.4a.

Procedure 1: DrawPoints

Require: Points D_{pt}, Point FBO F_{pt}
1: Initialize array A to 0
2: Clear point FBO F_{pt}
3: **for** each $p = (x, y) \in D_{pt}$ **do**
4: $(x', y') = transform(p)$
5: $F_{pt}(x', y')$ += 1 *% can be any function, see Section 3.5*
6: **end for**
7: **return** A, F_{pt}

Step II. Draw polygons: The second step renders all the polygons and incrementally updates the query result. As explained in Section 3.3, the polygons are first triangulated. All triangles corresponding to a polygon are assigned the same key (or ID) as that polygon. As before, the vertices of the polygons are transformed into the screen space and the rasterization converts the polygons into discrete fragments. The generated fragments are then processed in the fragment shader (Procedure 2). When processing a fragment corresponding to a polygon with ID i, the count of points corresponding to this pixel (stored in F_{pt}) is added to the result $A[i]$ corresponding to polygon i. Figure 3.4b highlights the pixels that are counted with respect to one of the polygons. After all polygons are rendered, the array A stores the result of the query. As we discuss in Section 3.5, this approach can be extended to handle other aggregation and filtering conditions.

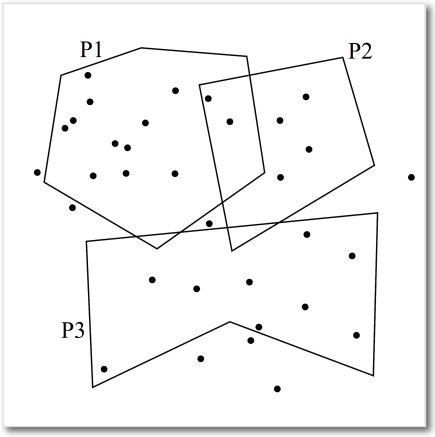


Figure 3.3 – Example input.

Procedure 2: DrawPolygons

Require: Polygon fragment (x', y'), Polygon ID i,
Point FBO F_{pt}, Array A
1: $A[i] = A[i] + F_{pt}(x', y')$ *% same function as in Procedure 1*
2: **return** A

3.4.2 Bounded Raster Join

Raster join is an approximate technique that introduces some false positive and false negative points. In this section, we show that the number of these errors depends on the resolution and their 2D location can be bounded.

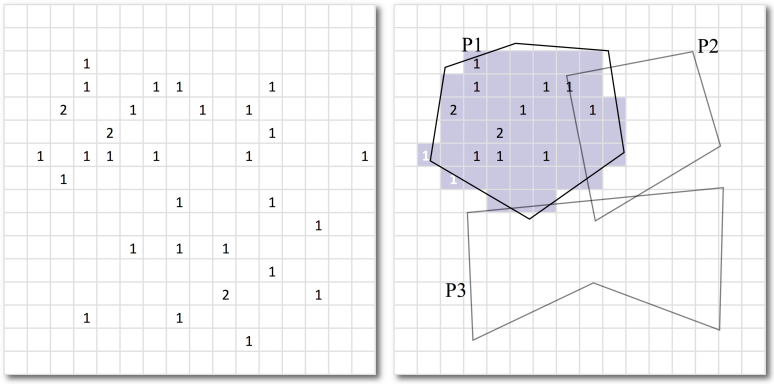


Figure 3.4 – The raster join approach first renders all points onto an FBO storing the count of points in each pixel (a). In the second step, it aggregates the pixel values corresponding to fragments of each polygon (b).

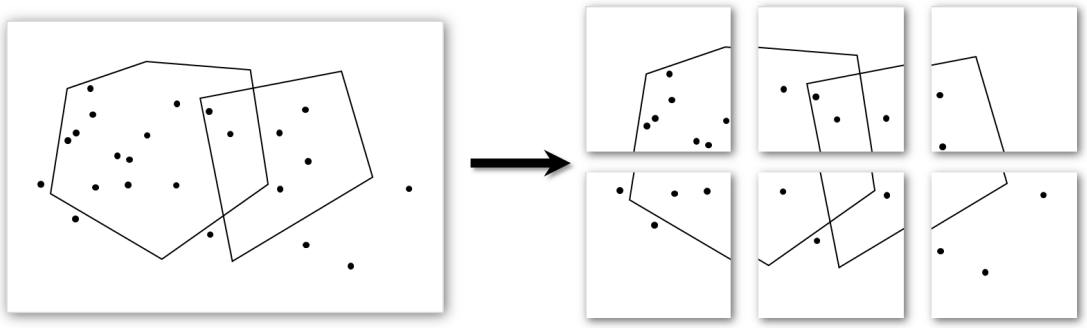


Figure 3.5 – When the resolution required to satisfy the given ϵ-bound is greater than what is supported by the GPU, the canvas used for drawing the geometries is split into multiple small canvases, each having resolution within the GPU's limit.

The introduction of *false negatives* is an artifact of the rasterization of the triangles that compose a polygon: a pixel is part of a triangle only when its center is inside the triangle. As a result, the points that fall in the intersection between a pixel and a triangle not containing the pixel's center are not aggregated. The pixels that intersect the polygon outline are considered to be part of the polygon and they introduce *false positives*, as all the points contained in those pixels are aggregated. In the example shown in Figure 3.4b, P1 is approximated by the violet fragments and the false positive counts are highlighted in white. By increasing the screen resolution, the pixel size decreases and thus pixels better approximate the polygon outline. As a result, the expected number of both false positives and false negatives decreases. Clearly, with an appropriately high resolution, we can converge to the actual aggregate result.

In real-world data, there is typically uncertainty associated with respect to the location of a point. Similarly, polygon boundaries (which often correspond to political boundaries) are fuzzy, in the

sense that there is often some leeway as to their exact location. For example, the neighborhood boundaries of NYC fall on streets, and in most cases, the whole street surface (rather than a one-dimensional line) is considered to be the boundary. This means that when analyzing data over neighborhoods, it is often admissible to consider data points falling on boundary streets to be part of either of the two adjacent neighborhoods. In such cases, it is sufficient to compute the aggregate with respect to a polygon i' that closely approximates the given polygon i. Formally, a polygon i' ϵ-approximates the polygon i if the Hausdorff distance $d_H(i, i')$ between the polygons is at most ϵ, where

$$d_H(i, i') = \max \left\{ \max_{p' \in i'} \min_{p \in i} d(p, p'), \max_{p \in i} \min_{p' \in i'} d(p', p) \right\}$$

Here, $d(p', p)$ denotes the Euclidean distance between two points.

Given ϵ, raster join can guarantee that $d_H(i, i') \leq \epsilon$, by using a pixel side length equal to $\epsilon' = \frac{\epsilon}{\sqrt{2}}$ (i.e., the length of the diagonal of the pixel is ϵ). Intuitively, this ensures that any false positive (false negative) point that is present (absent) in the approximate polygon, and thus considered (or not) in the aggregation, is within a distance ϵ from the boundaries of polygon i. For example, the outline of the violet pixelated polygon in Figure 3.4b represents the approximation used corresponding to P1. In the example of NYC neighborhoods, a meaningful aggregate result is obtained by using a pixel size approximately equal to the average street width.

The required resolution onto which the points and polygons are rendered to guarantee the ϵ-bound is $w' \times h' = \frac{w}{\epsilon'} \times \frac{h}{\epsilon'}$, where $w \times h$ are the dimensions of the bounding box of the polygon data set. When ϵ becomes small, the required resolution $w' \times h'$ can be higher than the maximum resolution supported by the GPU. To handle such cases, the canvas is split into smaller rectangular canvases, and the raster join algorithm described in Section 3.4.1 is executed over each one of them. This multi-rendering process is illustrated in Figure 3.5. Recall that during the rendering process, the points or polygons that do not fall onto the canvas are *automatically* clipped by the graphics pipeline. This ensures that every point-polygon pair satisfying the join is correctly counted exactly once.

Typically, in a visualization scenario such as the motivating example in Section 3.1, it is perfectly acceptable to trade-off accuracy for interactivity, and increase the query rate by performing only a single rendering operation with a relatively low resolution. For example, Figure 3.6(left) shows the number of taxi pick-ups that happened in the month of June 2012 over the neighborhoods of NYC as obtained using the raster technique with a canvas resolution of approximately $4k \times 4k$ that corresponds to $\epsilon = 20$ meters. Note that this approximate result is almost indistinguishable from the visualization obtained from an accurate aggregation (right), but it can be computed at a fraction of the time. Also, if we fix a resolution as is common in visualization interfaces, when the user zooms into an area of interest, a smaller region is rendered with a larger number of pixels. Effectively, this is equivalent to computing the aggregation with a higher accuracy without any

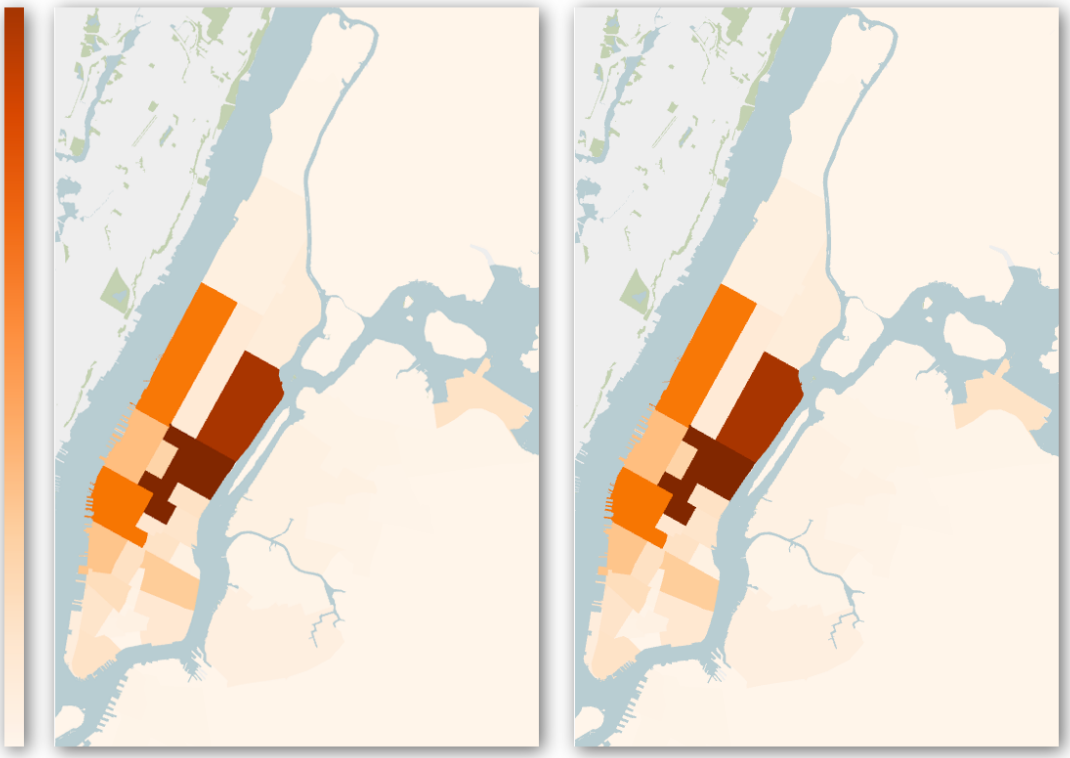


Figure 3.6 – Visualizing the approximate (left) and accurate (right) results of the example query in Figure 3.1. The ϵ-bound was set to 20m. Note that the two visualizations are virtually indistinguishable from one another.

significant change in computation times (since the FBO resolution does not change). Thus, our approach is naturally suited for LOD rendering.

3.4.3 Accurate Raster Join

While Bounded Raster Join derives good (and bounded) approximations for spatial aggregation queries, some applications require accurate results. In this section, we describe a modification of the core raster approach that obtains exact results through the addition of a minimal number of PIP tests.

Consider the same point and polygon data sets described in the previous section, but as illustrated in Figure 3.7a. Notice that certain fragments (pixels), colored green and white respectively, are either completely inside one of the polygons, or outside all polygons. Grid cells colored violet are on the boundary of one or more polygons. Recall that the errors from the raster approach are due only to the points that lie in these boundary pixels. This observation can be used to minimize the number of PIP tests: by performing tests just on these points, we can guarantee that no errors occur. This is accomplished in three steps.

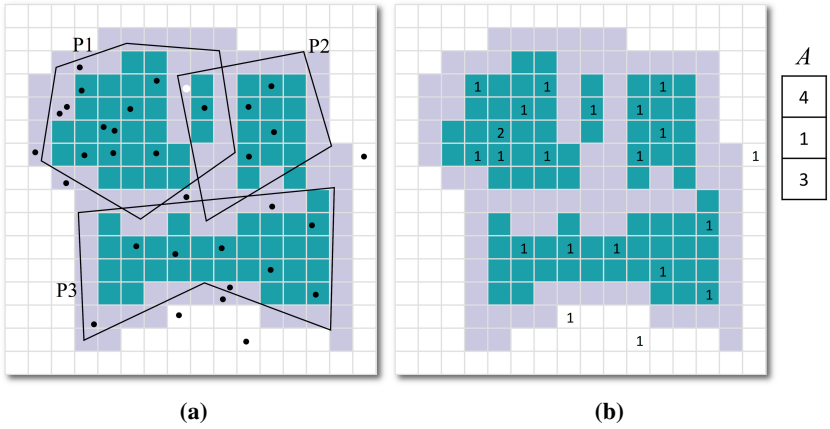


Figure 3.7 – Accurate raster join performs PIP tests only on points that fall on the violet cells in (a) that correspond to pixels forming the boundaries of the polygons. The other points are accumulated in the green pixels (b), which are then added to the polygons that are "drawn" over them.

1. Draw the outline of all the polygons: In this step, the boundaries of the set of polygons are rendered onto an FBO. In particular, the fragment shader assigns a predetermined color to the fragments corresponding to the boundaries of the polygons. The FBO is first cleared to have no color ($[0, 0, 0, 0]$), thus ensuring that at the end of this step, only pixels on the boundary will have a color. The outline FBO for the example data will consist of an image having only the violet pixels from Figure 3.7.

2. Draw points: This step (Procedure 3) builds on the core raster approach described above. As before, we maintain a result array A initialized to 0. When a point is processed, it is first transformed into the screen space. If the fragment corresponding to the point falls into a boundary pixel (which is determined by examining the pixel color in the Boundary FBO), the point is processed with Procedure 4.

Procedure 3: AccuratePoints

Require: Polygon Index *Ind*, Points D_{pt}, Boundary FBO F_b
1: Initialize array A to 0
2: Clear point FBO F_{pt}
3: **for** each $p = (x, y) \in D_{pt}$ **do**
4: $(x', y') = transform(p)$
5: **if** $F_b(x', y')$ is a boundary **then** *% test pixel color in FBO*
6: *execute* Procedure JoinPoint(*Ind*, p, A)
7: **else**
8: $F_{pt}(x', y')$ += 1 *% same function as in Procedure 1*
9: **end if**
10: **end for**
11: **return** A, F_{pt}

Procedure 4: JoinPoint

Require: Polygon Index *Ind*, Point (x, y), Array A
1: P = *Ind*.query(x, y)
2: **for** each $r_i \in P$ **do**
3: **if** r_i contains p **then**
4: $A[i] = A[i] + 1$ *% same function as in Procedure 1*
5: **end if**
6: **end for**
7: **return** A

This procedure first uses an index over the polygons to identify candidate polygons that might contain the point, and then performs a PIP test for every candidate. Since our focus is on time efficiency, we use a grid index that stores in each grid cell the list of polygons intersecting it, thus allowing for constant $O(1)$ lookup time. If a point is inside the polygons with IDs $I = \{i_1, i_2, \ldots, i_l\}$, $l \leq k$, where k is the total number of polygons, then each of the array elements $A[i]$, $i \in I$, is incremented by 1.

If the fragment does not correspond to a boundary pixel, then this fragment is rendered onto a second FBO. In this FBO, as in the core approach, we use the color channels of a pixel to store the count of points falling in that pixel. This step results in two outputs: A, which stores the partial query result corresponding to data points that fall on the boundary of the polygons; and F_{pt}, an FBO storing the count of points that fall into each of its pixels (see Figure 3.7b).

3. Render polygons: The final step simply renders all the polygons and updates the query result when processing the polygon fragments in the fragment shader (Procedure 5). The only difference from the core approach in this procedure is checking if a fragment corresponding to a polygon with ID i falls on a boundary pixel. If the fragment is on a boundary pixel, then it is discarded since all points falling into that pixel have already been processed in the previous step. Otherwise, all points that fall into the pixel are *inside* this polygon. Thus, the count of points corresponding to the pixel (stored in F_{pt}) is added to the result $A[i]$ corresponding to polygon i. Note that when polygons intersect, fragments completely inside one polygon can be on the boundary of another polygon. The white point in Figure 3.7a is one such example: it lies inside P1, but on the boundary of P2. After all polygons are rendered, the array A stores the result of the query.

Procedure 5: AccuratePolygons

Require: Polygon fragment (x', y'), Polygon ID i,
Boundary FBO F_b, Point FBO F_{pt}, Array A
1: **if** $F_b(x', y')$ is **not** a boundary **then** *% test FBO pixel color*
2: $A[i] = A[i] + F_{pt}(x', y')$ *% same function as in Procedure 4*
3: **end if**
4: **return** A

3.5 Raster Join Extensions

In this section, we discuss how our approach can be extended to handle different aggregations and filtering clauses, as well as data larger than GPU memory. We also describe how accurate ranges can be computed for the aggregate results. While the bounded approach provides guarantees with respect to the spatial region to take into account the uncertainties in the spatial data, providing bounds over the query result can be also useful for a more in-depth analysis.

Aggregates. Aggregate functions are categorized into distributive, algebraic and holistic [62]. *Distributive aggregates*, such as count, (weighted) sum, minimum and maximum, can be computed by dividing the input into disjoint sets, aggregating each set separately and then obtaining the final result by further aggregating the *partial aggregates*. *Algebraic aggregates* can be computed by combining a constant number of distributive aggregates, e.g., average is computed as *sum*/*count*. *Holistic aggregates*, such as median, cannot be computed by partitioning the input. The description in this chapter focuses on *count* queries, while our current implementation also supports *sum* and *average*. However, note that our solutions apply to any distributive or algebraic (but not to holistic) aggregates in a straightforward manner. When computing the average, as with the count function, one of the color channels in the FBO (e.g., red) is used for counting the number of points while another channel (e.g., green) is used to sum the appropriate attribute. Similarly, instead of a single output array A, we use two arrays A_1 and A_2 to maintain the sum and count values when the polygons are processed (or when PIP tests are performed in the accurate variant). After all polygons are drawn in the final step of the algorithm, the query result is obtained by dividing the elements of the sum array A_1 by the elements of the count array A_2. Note that the data corresponding to the aggregated attribute is also transferred to the GPU.

Query Parameters. When query constraints are specified, they can also be tested on the GPU for each data point. The constraint test is performed in the vertex shader before transforming the point into screen space. The vertex shader discards the points that do not satisfy the constraint by positioning them outside the screen space so that they are clipped and they are not further processed in the fragment shader. We currently support the following constraints: $>, \geq, <, \leq$, and $=$. Note that the data corresponding to the attributes over which constraints are imposed is also transferred to the GPU.

Out-of-Core Processing. When the data points do not fit into GPU memory, they are split into batches that fit into the GPU. Then the query is executed on each of the batches and the results are combined. Thus, a given point data set has to be transferred to the GPU *exactly once*. Current generation GPUs have at least a few GB of memory that can easily fit several million polygons (depending on their size). Thus, we assume that the polygon data set fits into GPU memory and does not need to be transferred in batches.

Estimating the Result Range. We extend the bounded variant to compute a range for the aggregate result at each polygon. This is accomplished using the boundary pixels corresponding to the polygons as follows. Given a polygon i, let P_i^+ (P_i^-) be the set of pixels on its boundary that

contain false positive (negative) results. Since only these pixels contribute to the approximation, counting the points contained in them provides loose bounds on the result range. In particular, the sums $\epsilon_i^+ = \sum_{(x,y)\in P_i^+} F_{pt}(x,y)$ and $\epsilon_i^- = \sum_{(x,y)\in P_i^-} F_{pt}(x,y)$ are used to compute the worst case lower and upper bounds respectively, resulting in the interval $[A[i] - \epsilon_i^+, A[i] + \epsilon_i^-]$ with 100% confidence.

Independent of the actual data distribution, since the region corresponding to a pixel covers a very small fraction of the spatial domain, we can reasonably assume that the spatial and value-domain distribution of the data points *within each pixel is uniform*. Under this assumption, we provide tighter expected intervals by computing the intersection between the boundary pixels and the polygons. In particular, let $f_i(x,y)$ denote the fraction area of the pixel (x,y) that intersects polygon i. Then the expected lower and upper bounds, respectively, are computed as before using:

$$\epsilon_i^+ = \sum_{(x,y)\in P_i^+} f_i(x,y) \times F_{pt}(x,y)$$
$$\epsilon_i^- = \sum_{(x,y)\in P_i^-} f_i(x,y) \times F_{pt}(x,y)$$

The corresponding intervals for *sum* and *average* can be computed in a similar fashion.

3.6 Implementation

In this section we first discuss the implementation of the raster join approaches using OpenGL[3]. We then briefly describe the GPU baseline used for the experiments.

3.6.1 OpenGL Implementation

We used C++ and OpenGL for the implementation. We make extensive use of the newer OpenGL features, such as *compute shaders* and *shader storage buffer objects* (SSBO). Compute shaders add the flexibility to perform general purpose computations (similar to cuda [112]) from within the OpenGL context while making the implementation (hardware) manufacturer independent. SSBOs enable shaders to write to external buffers (in addition to FBOs). For memory transfer between the CPU and GPU, we use the newly introduced *persistent mapping* that is part of OpenGL's techniques for Approaching Zero Driver Overhead.

[3] https://github.com/vida-nyu/raster-join

Polygon Triangulation. To triangulate polygons, we use the *clip2tri* library [38], which implements an efficient constrained Delaunay-based triangulation strategy. Triangulation is accomplished in parallel on the CPU and the set of triangles is transferred to the GPU during query execution.

Bounded Raster Join. Each of the two steps of the bounded approach, i.e., drawing points followed by drawing polygons, is composed of two shaders – a vertex shader and a fragment shader.

When drawing points, we transfer them to the GPU by copying them to a persistently mapped buffer that is used as a vertex buffer object (VBO). Each vertex shader instance takes a single data point from the VBO and transforms it into screen space (Line 4 in Procedure 1). The transformed point is processed in the fragment shader, which essentially updates the FBO at the given location (Line 5 in Procedure 1). Note that the memory for the FBO is allocated directly on the GPU.

When drawing polygons, the triangle coordinates are passed to the GPU as part of the VBO, and the vertex shader again transforms the endpoints to screen space. The rasterization is accomplished as part of the OpenGL pipeline, and each fragment resulting from this operation is processed in the fragment shader (Procedure 2). Since the FBO from the previous step is already on the GPU, it is simply bound as a texture to the fragment shader, thus ensuring there is no unnecessary data transfer between the CPU and GPU. The result array A is maintained as an SSBO, and atomic operations are used when updating it. An advantage of SSBOs is that they allow processing intersecting polygons in a single pass thus avoiding unnecessary, additional processing.

Computing Result Ranges. Recall that the boundary pixels that contribute to both false positives and false negatives have to be identified to compute the result intervals. False positive pixels are identified by simply drawing the outline of a polygon. Identifying false negative pixels, however, is less straightforward. To accomplish this, we use *conservative rasterization* that allows rendering *all partially covered* pixels: the outline is drawn using conservative rasterization, and pixels that are not part of the regular rasterization form the false negative pixels. Conservative rasterization is supported via a custom OpenGL extension (*GL_NV_conservative_raster*) on Nvidia GPUs. On non-Nvidia GPUs, conservative rasterization can be accomplished by drawing a thicker outline and discarding pixels that do not intersect with the drawn polygon.

Deriving the tighter expected result interval requires computing the intersection between a pixel and its corresponding polygon. To do this efficiently, in the vertex shader, in addition to the transformed coordinates, we output the edge that is being drawn. We then use the Cohen-Sutherland line clipping algorithm [52] in the fragment shader to compute the fraction of the pixel that intersects with the polygon.

Accurate Raster Join. The first step of the accurate variant (drawing polygon boundaries) is again implemented as a vertex and fragment shader, where the boundaries are stored in an FBO. Conservative rasterization is used to ensure that no boundary pixels are missed. The second step

(lines 4–9 in Procedure 3) is implemented using compute shaders. As before, persistent mapped buffers are used for sharing data between the CPU and GPU. In addition to the data points, this step also requires a grid index on the query polygons. This index is created on-the-fly on the GPU as described next. The implementation of the third step (Procedure 5) is similar to that of the bounded raster join.

Polygon Index. We implemented a grid index that stores a polygon identifier in all the grid cells intersecting the bounding box of that polygon. The grid is represented as an array of linked lists, one for every grid cell. Each linked list stores all polygons that are assigned to a grid cell. We build the index on the GPU on-the-fly for every query in two passes. Given a grid resolution, the first pass computes the number of cells each polygon intersects with to estimate the size of the index. The second pass assigns the polygons to their corresponding cells. Since dynamic memory allocation is not supported on the GPU, we allocate the required memory directly on the GPU as a single contiguous region and implement a custom linked list. This memory is discarded after the query is executed.

Query Options. We chose to pass attributes as part of the VBO rather than regular buffers to allow for an efficient stratified access to the data when processing the vertex information in the vertex shader. However, this imposes the restriction that the size of each vertex must be fixed at compile time. As a result, in our implementation, we support constraints (which are conjunctions) over a maximum of 5 attributes. We can increase this constant up to the hardware limit in the shader code.

3.6.2 Baseline: Index Join Approach

As we show in the next section, using current GPU-based spatial join techniques [175] to execute the spatial aggregation is not very efficient mainly due to the materialization of the spatial join prior to the aggregation. To have a better baseline to compare our rasterization-based approaches, we extend existing index-based techniques to combine the spatial join operation with the aggregation so as not to explicitly materialize the join.

The key idea is to use an index on the polygon data to identify polygons on which to perform PIP tests. As with the accurate raster join, we use the grid index for this purpose. The query is executed as shown in Procedure 6. As with the raster join variants, the result array A is initialized to 0. Then the algorithm processes each point $p(x, y) \in D_{pt}$ using Procedure 4. Note that in the

Procedure 6: IndexJoin

Require: Polygon Index *Ind*, Points D_{pt}
1: Initialize array A to 0
2: **for** each $p = (x, y) \in D_{pt}$ **do** *% Can be run in parallel*
3: *execute* Procedure JoinPoint(*Ind*, p, A)
4: **end for**
5: **return** A

Table 3.1 – Polygonal data sets and processing costs.

Region	Nr of polygons	Text file size	Triangulation	Index Creation		
				GPU	Multi-CPU	Single-CPU
NYC neighborhoods	260	877 KB	20 ms	10 ms	0.57 s	2.15 s
US counties	3945	20 MB	0.66 s	14 ms	23.34 s	37.05 s

case of accurate raster join, this Procedure is executed only for a *subset* of points that are within a small distance from the polygon boundaries. After all points are processed, the array A contains the query result. Similar to the accurate raster join variant, the above query is implemented using a compute shader.

3.7 Experimental Evaluation

In this section, we first describe the experimental setup and then present a thorough experimental analysis that demonstrates the benefits of our raster join approach using two real-world data sets. Sections 3.7.3–3.7.5 discuss the scalability of the approaches when data fits in main memory. The goal of these experiments is threefold: 1) demonstrate the benefits of exploiting the parallelism of GPU; 2) verify the scalability of the approaches with increasing input sizes; and 3) show that the bounded variant outperforms all the other approaches in terms of query performance. Section 3.7.6 provides an in-depth analysis of the accuracy trade-offs of the bounded raster join approach. Finally, Section 3.7.7 presents experiments on data sizes larger than the main memory.

3.7.1 Experimental Setup

Hardware Configuration. The experiments were performed on a Windows *laptop* equipped with an Intel Core i7 Quad-Core processor, running at 2.8 GHz, 16 GB RAM and 1 TB SSD, and an NVIDIA GTX 1060 mobile GPU with 6 GB of GPU memory.

Data Sets. We use two real-world data sets for our experiments: NYC yellow taxi and Twitter. The *NYC Taxi data set* contains details of over 868 million yellow taxi trips that occurred in NYC over a 5 year period (2009 - 2013). The data is available [145] as a collection of csv files, which when converted to binary occupy 72 GB. Each record corresponds to a taxi trip and consists of two spatial attributes (pickup and drop-off locations), two temporal attributes (pickup and drop-off times), as well as other attributes such as fare, tip, and number of passengers. The data is stored as columns on disk and the required columns (attributes) from the Taxi data set are loaded into main memory prior to performing any experiments. The *Twitter data set* was collected from Twitter's live public feed over a period of 5 years. It consists of over 2.29 billion geo-tagged tweets in the

USA formatted as json records. Each tweet record has attributes corresponding to the location and time of the tweet, the actual text, and other information such as the favorite and retweet counts. When converted to binary, the data (excluding the text) occupies 69 GB on disk. Note that both data sets are *skewed*. Taxi trips are mostly concentrated in Lower Manhattan, Midtown, and airports, while there is a denser concentration of tweets around large cities. Sections 3.7.2–3.7.6 use the taxi data set. To perform the join queries, we also use two *polygon data sets*, summarized in Table 3.1. These data sets contain complex polygons commonly used by domain experts in their analyses.

Queries. For the experimental evaluation, we use *Count()* as the most representative aggregate function. Unless otherwise stated, no filtering is performed on additional attributes. To vary the input sizes, we first divided the data into roughly equal time intervals. The input size of a query was then increased by using data from increasing number of time intervals.

CPU Baseline: Index Join Approach. In addition to the GPU approaches, we also implemented the Index Join Approach on the CPU (described in Section 3.6). We further optimized the approach by assigning a polygon only to those grid cells that the actual geometry intersects. That is, we build the polygon index by first identifying all the cells intersecting with the MBR of the polygon, and then perform cell-polygon intersection tests. The algorithm was implemented in C++. We also implemented a parallel version with OpenMP, where we used *#pragma omp parallel for* to parallelize the PIP tests (Line 2 in Procedure 6). To avoid locking delays, each thread maintains the aggregates in a thread-local data structure, and all the aggregates are merged into a single result array in the end. The building of the polygon index was also parallelized (each polygon was processed independently).

Processing Polygon Data. Recall that both rasterization variants require the polygons to be triangulated (Section 3.3), while the accurate variant, and the CPU as well as GPU baselines require the creation of an index. For all the GPU approaches, our implementation computes the triangulation (in parallel on the CPU) and the indexes (on the GPU) on-the-fly for each query. On the other hand, since the CPU computation is much slower, the indexes were pre-computed. Table 3.1 shows the time taken for each of these cases. To be consistent, we do not include the polygon processing time in the reported query execution time. However, note that even if these time were included, they would have a minimal effect on the performance of the GPU approaches.

Configuration Parameters. We limited the GPU memory usage to 3 GB, and the maximum FBO resolution to 8192×8192. Unless otherwise stated, the default ϵ-bound for NYC polygons is 10 m, and 1 km for US polygons. The resolution of the grid index for the neighborhood polygonal data set was set to 1024^2. For US counties, the GPU approaches use a grid index with 1024^2 cells, while the CPU baselines use 4096^2 cells. The index resolution for the GPU approaches was chosen based on the total time, including index creation time, since this was part of the query execution. The overall performance of using a 1024^2 index far outweighed that of using a 4096^2 index on the GPU. On the other hand, since we pre-computed the index for the CPU implementation, we chose the resolution that provided best query performance.

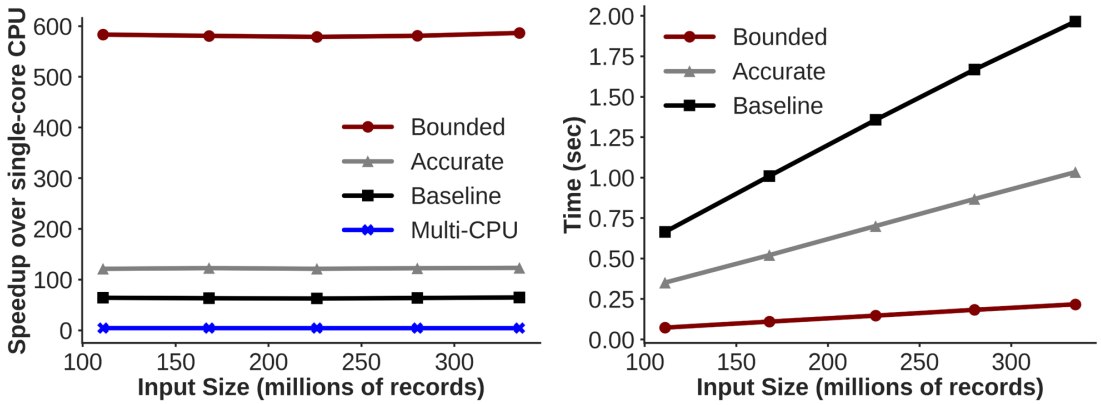


Figure 3.8 – Scaling with increasing input sizes for Taxi ⋈ Neighborhood when the data fits in GPU memory. (Left) Speedup over single-CPU. (Right) Total query time. Bounded Raster Join has the best scalability as it eliminates all PIP tests. Accurate Raster Join performs fewer PIP tests than the Baseline.

3.7.2 Choice of GPU Baseline

Table 3.2 compares our GPU index-based approach (Index Join) with state-of-the-art work on GPU join/aggregation[4] [175]. Our implementation performs 2-3× faster, mainly due to avoiding the materialization of the join result. We could not perform experiments with bigger input sizes as the provided code ran out of GPU memory. In the remaining experiments, given its clear advantages, we use our Index Join as the GPU baseline.

Table 3.2 – Choice of GPU baseline.

Input Size (#points)	Zhang et al. [175] (total time - ms)	Index Join Baseline (total time - ms)
57,676,723	1060	344
111,659,661	1649	651
168,368,285	2129	999

3.7.3 Scalability with Points

In-Memory Performance. Figure 3.8 (left) plots the *speedup* of the parallel approaches (GPU and CPU) relative to our single-threaded CPU baseline when the point data sets fit in the GPU memory (i.e., the GPU memory holds the entire data set and data need not be transferred from the CPU to the GPU). Figure 3.8 (right) plots the *total time* against input size. The rasterization-based approaches are over two orders of magnitude faster than the single-core CPU implementation. Moreover, *the bounded variant is over 4 times faster than the accurate versions*. Given that

[4]Code made available by the authors at http://geoteci.engr.ccny.cuny.edu/zs_supplement/zs_gbif.html

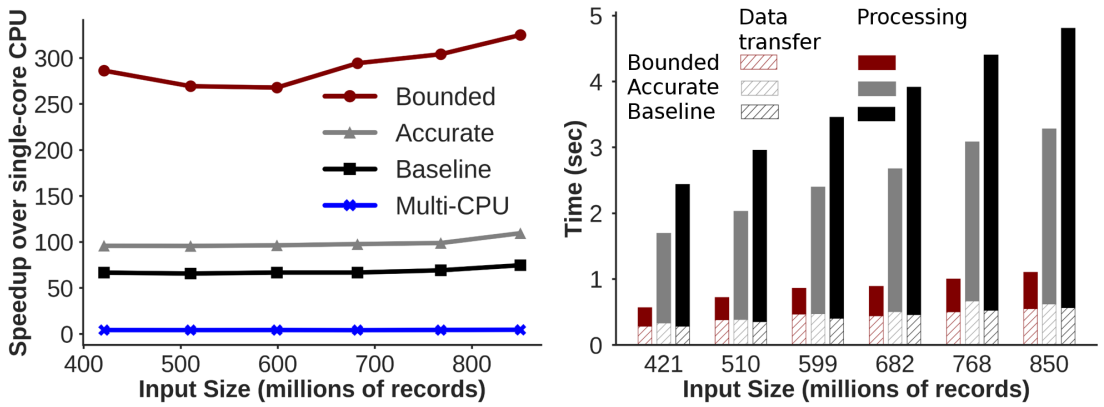


Figure 3.9 – Scaling with increasing input sizes for Taxi ⋈ Neighborhood when the data does not fit in GPU memory. (Left) Speedup over single-CPU. (Right) Break down of the execution time. Note that the memory transfer between CPU and GPU dominates the execution time for the bounded approach.

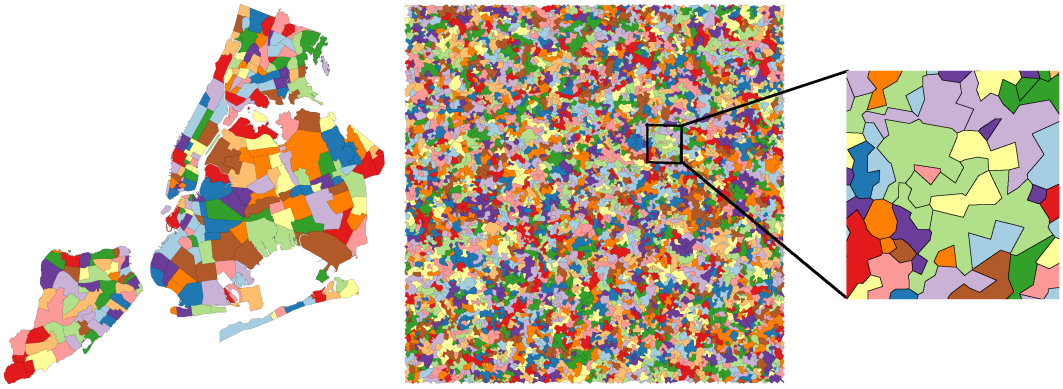


Figure 3.10 – [Best viewed in color] Neighborhoods of New York City (left). 4096 synthetic polygons generated over the same area (right).

our test system has a quad-core processor (with a total of 8 threads), the multi-core CPU implementation provides a 5× speedup over the single-core CPU implementation. Thus, for the same laptop the GPU offers at least an order of magnitude more parallelism.

Out-of-Core Performance. While significant speedups are obtained for in-memory queries, large speedups are achieved even when the data does not fit in GPU memory. As illustrated in Figure 3.9, the GPU approaches still obtain over an order of magnitude speedup over the CPU implementation while bounded raster join has a speedup of over two orders of magnitude. Note that, since the query times in this case are in milliseconds, the speedups are affected by even small fluctuations in the time (e.g., due to other windows background processes). The scalability observed for the different approaches is similar to that when data fits in GPU memory. By eliminating the costly polygon containment tests, the bounded approach significantly outperforms the other approaches. Even when the input size is around 868 million points, query execution

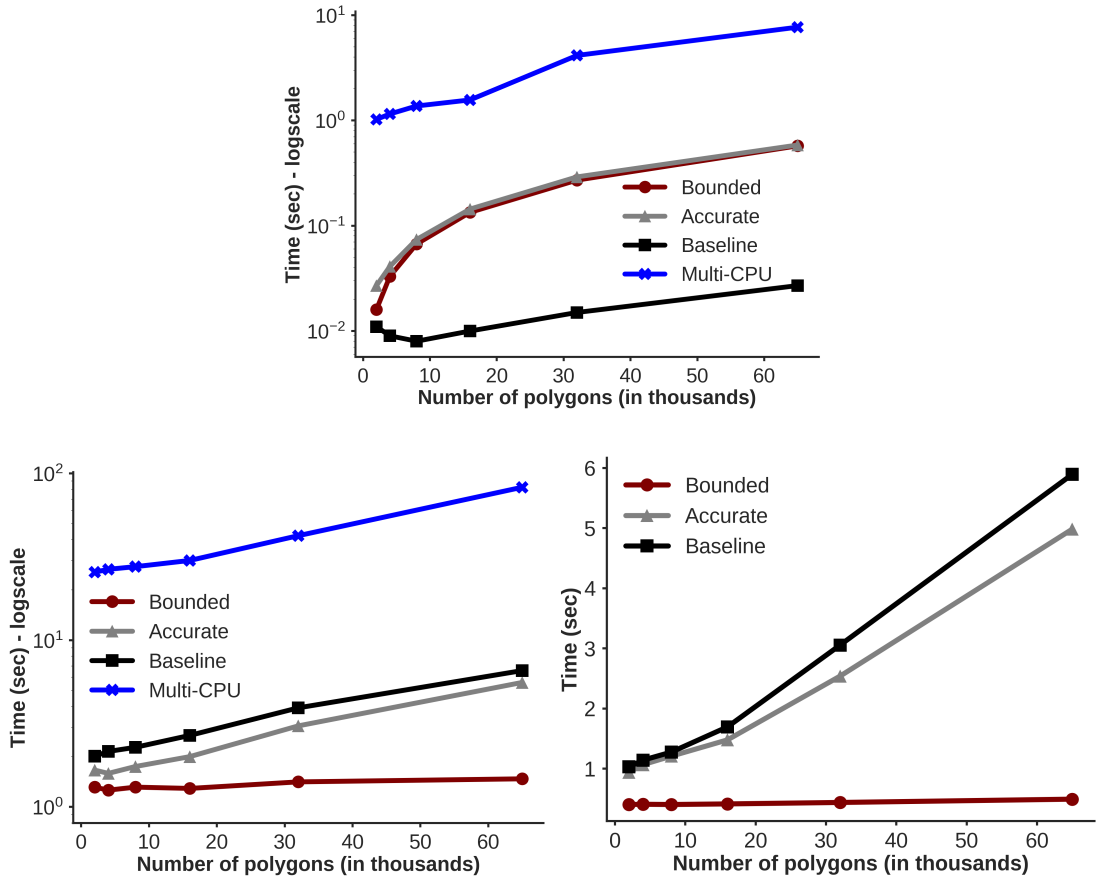


Figure 3.11 – Scaling with polygons. (Top) Polygon processing costs. (Bottom Left) Total query time when data does not fit in GPU memory. (Bottom Right) GPU processing time. Note that increasing the number of polygons has almost no effect on Bounded Raster Join.

takes only 1.1 seconds. The linear scaling also shows that the computation time is typically not affected by the number of additional passes required in the out-of-core scenario. To better understand the reduced speedup attained by the out-of-core technique compared to in-memory, we broke down the total execution time into the different components of query evaluation, i.e., processing and data transfer. The data transfer has a significant contribution in the overall time, especially in the case of bounded raster join where it dominates the execution time.

3.7.4 Scalability with Polygons

Generating Polygons. Since real-world polygonal data consists of a small number of polygons (100's to 1000's), we generated polygonal data to test the scalability of our approaches with the number of polygons. Real-world polygonal data sets have a combination of simple as well as complex-shaped (concave and arbitrary) polygons with varying sizes, as seen in Figure 3.10 (left). Our goal was to generate synthetic polygons with properties similar to the real ones. To accomplish this, we use the Voronoi diagram to generate a collection of convex polygons of varying

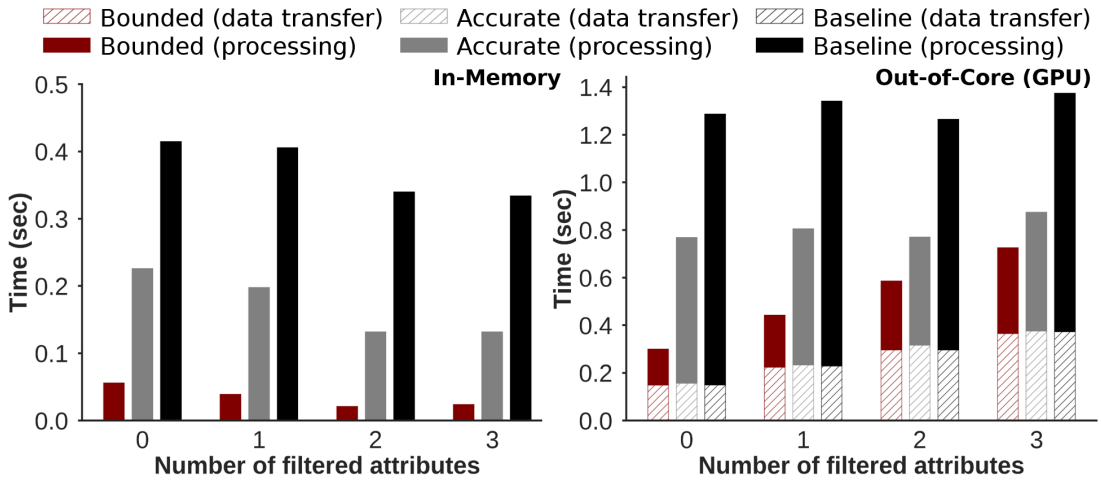


Figure 3.12 – Scaling with number of attribute constraints.

sizes (based on the location of the points) and then ensure that concave and more complex shapes are generated by merging multiple adjacent convex polygons. More specifically, to generate n polygons, we first randomly generated $4n$ points within the rectangular extent of the data. We then computed the constrained Voronoi diagram over these points. This generates a collection of $4n$ convex polygons partitioning the rectangular region. Next, we randomly chose two neighboring polygons and merged them into a single polygon. We repeated this step until only n polygons remained. While on average 4 polygons are merged into one, there are cases where a higher number of polygons are merged (creating complex shapes), as well as cases where the simpler convex shapes are retained, thus mimicking the real-world scenario. The generated polygons are shown in Figure 3.10 (right).

Polygon Processing Costs. Figure 3.11 (top) shows the cost of processing the polygons (i.e., triangulation and index creation). As with the neighborhood data, we build a grid index with 1024^2 cells. Recall that the bounded variant requires only triangulation, the baseline only grid index creation, and the accurate variant both. As expected, triangulation time increases with an increasing number of polygons. When building the index, since the polygons partition the space, we touch all cells of the grid index one or more times depending on the polygons' structure. That explains the small drop at the beginning of the plot: even though the number of polygons increases, the sizes of their bounding boxes become smaller and each grid cell needs to be processed fewer times. As the number of polygons further increases, each grid cell intersects more polygon bounding boxes and needs to be processed more times thus increasing the building time. Note that even 64K polygons are processed in milliseconds. Thus, in dynamic settings where the polygons are not known a priori, they can be efficiently processed on-the-fly.

Performance. Figure 3.11 (bottom left) plots the total time when joining with 600 M points that do not fit in GPU memory. Figure 3.11 (bottom right) focuses on the time spent on the GPU. The performance gap between the accurate variant and the baseline is much smaller in this scenario.

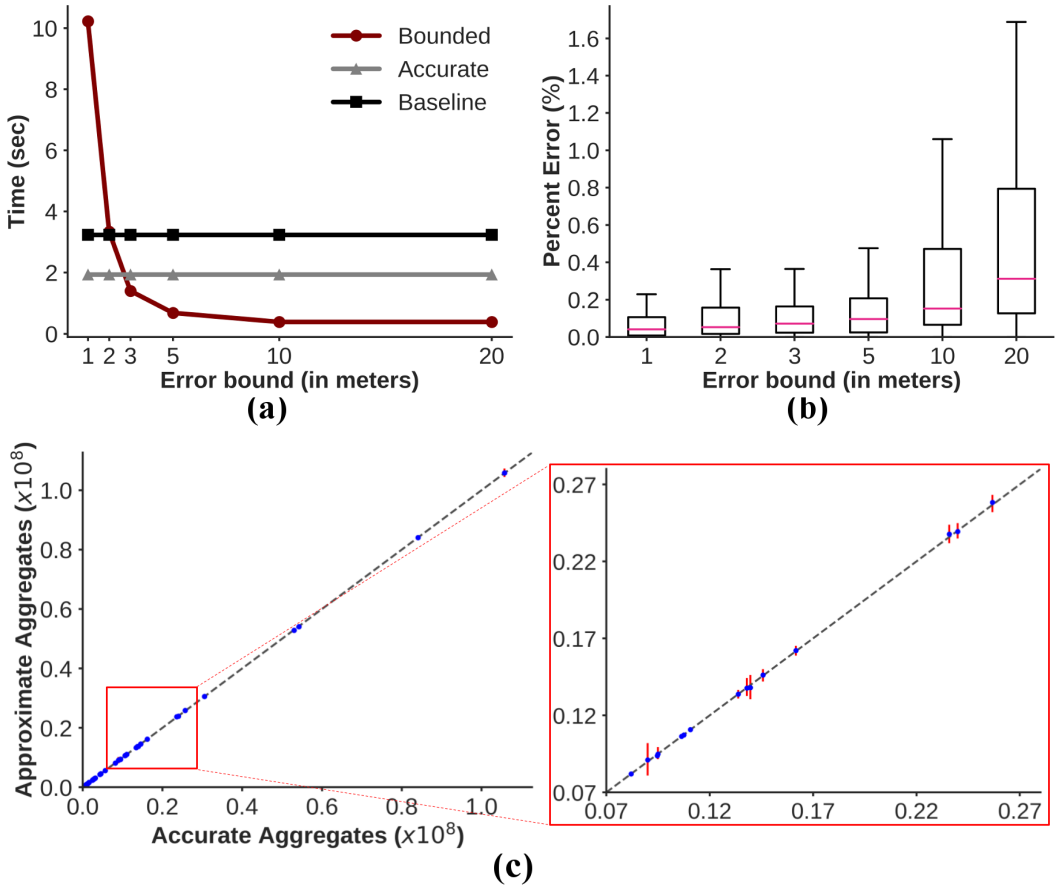


Figure 3.13 – Accuracy analysis. (a) Accuracy-time trade-off. (b) Accuracy-ϵ-bound trade-off. The box plot shows the distribution of the percent error over the polygons for different ϵ-bounds. (c) The scatter plot shows, for each polygon, the accurate value against the approximate value for $\epsilon = 20$ m. The red error bars indicate the expected result intervals (see the enlarged highlighted region).

Given the large number of polygons, the polygon outlines cover a significantly higher number of pixels, thus requiring more PIP tests to be performed. In the worst case, if the polygonal data set is very dense, the accurate variant degenerates into the baseline. Current generation GPUs can handle millions of polygons at fast frame rates. Since the bounded variant decouples the processing of points and polygons, increasing the number of polygons has almost no effect on the query time. The performance of the in-memory scenario is similar to that of Figure 3.11 (bottom right), which shows the GPU processing times.

3.7.5 Adding Constraints

As mentioned in Section 3.1, users commonly change query parameters interactively as they explore a data set. To test the efficiency of our approach in this scenario, we incrementally apply constraints on different attributes of the taxi data. Figure 3.12 shows the total execution time of

these queries for two input sizes, the first fitting in GPU memory (85 M points) and the second not (226 M points). The out-of-GPU-core breakdown shows that increasing the number of constraints increases the memory transfer time, as more data corresponding to the filtered attributes has to be transferred. However, the processing time is sometimes reduced with a higher number of constraints, because points that do not satisfy the constraints are discarded in the vertex shader, before performing any processing, thus reducing the amount of work done by the GPU.

3.7.6 Accuracy

Accuracy-Time Trade-Off. Figure 3.13(a) plots the trade-off between accuracy and query time for a query involving 600 M points (out-of-core). As the value of ϵ decreases, the number of rendering passes increases quadratically, thus the query time increases. After some point, the bounded variant becomes slower than the accurate. Analyzing this trade-off can help a query optimizer to automatically select a variant based on the value of ϵ.

Accuracy-ϵ-Bound Trade-Off. Figure 3.13(b) shows the effect of the specified ϵ-bound on the accuracy of the query results. The whiskers of the box plot represent the extent that is within 1.5 times the interquartile range of the 1st and 3rd quartiles, respectively. Decreasing the ϵ-bound decreases the error range converging towards the accurate values. The error range for the default value of $\epsilon = 10$ m is small, with a median of only about 0.15%. To show the actual differences in the aggregation results, we also plot the accurate vs. the approximate value for each of the polygons, using the coarsest bound ($\epsilon = 20$ m) in Figure 3.13(c). The fact that all the points lie very close to the diagonal indicates that even for a coarse bound a very good approximation is obtained. The bars in these plots denote the expected result interval that is computed (being very small, it is not clearly visible on the complete scatter plot). As seen in the highlighted region, our approach provides a tight interval even for a coarse ϵ value. The overhead of computing the intervals is negligible; computing them even for the costliest bound of $\epsilon = 1$ m required only an additional 140 ms. The accuracy trade-offs of the in-memory setup have a similar behavior.

Effect on Visualizations. Figure 3.6 shows side-by-side the visualizations computed through the bounded and accurate variations respectively. Note that the two visualizations are *perceptually* indistinguishable. The quality of the approximations can also be formally verified using *just-noticeable difference* (JND), a quantity used to measure how much a color should “increase" or “decrease" before humans can reliably detect the change [39]. In particular, sequential color maps used in the above visualizations can have a maximum of 9 perceivable color classes [68], resulting in a JND equal to $\frac{1}{9}$. A human can perceive the difference between the approximate and accurate visualizations, only when the difference between the corresponding normalized values is greater than $\frac{1}{9}$. However, the *maximum absolute error* between the normalized values even for the coarsest error bound ($\epsilon = 20$ m) is less than $0.002 \ll 0.11$, clearly showing that the difference from the visualization obtained using the bounded variation is not perceivable.

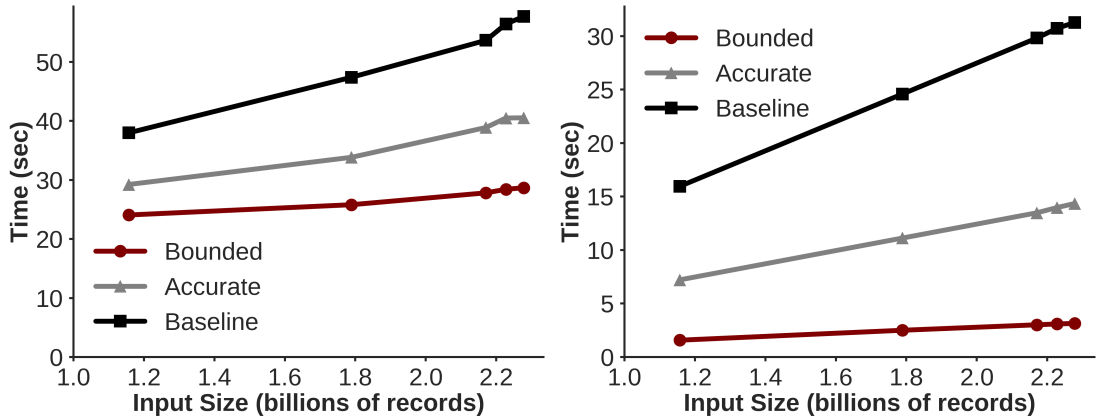


Figure 3.14 – Scaling with points when data does not fit in main memory (Twitter ⋈ County). (Left) Total query time. (Right) Processing time excluding memory access time.

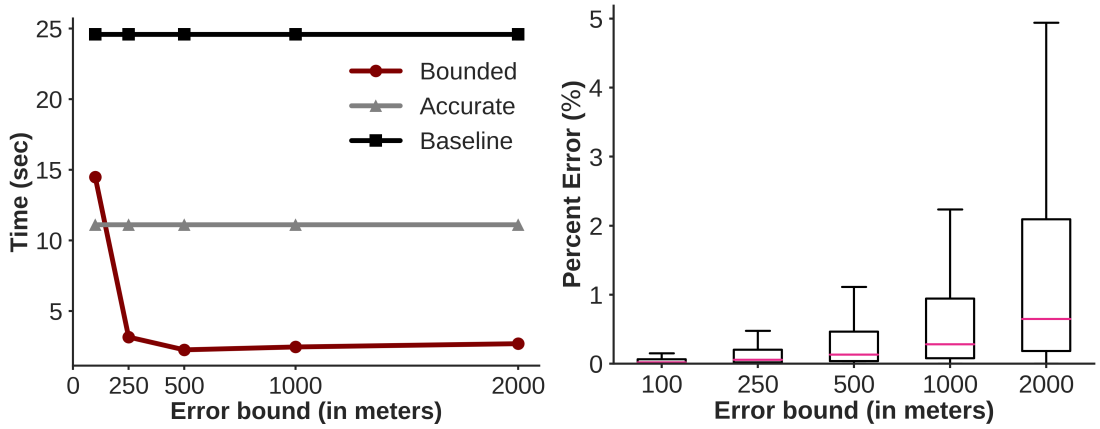


Figure 3.15 – Accuracy-Time trade-off (left) and ϵ-bound trade-off (right) using the Twitter data.

3.7.7 Performance on Disk-Resident Data

Figure 3.14 shows the performance of the different approaches when the data does not fit into the main memory of the system. We use the twitter data for this purpose, and aggregate over all counties in the USA. The increase in query time is primarily due to disk access times. Our implementation simply reads data from disk as and when required to transfer to the GPU, and does not apply any I/O optimizations such as parallel prefetching, which are beyond the scope of this work. Our focus was on the design of an efficient *operator* to perform spatial aggregation that can be integrated into any existing DBMS which efficiently handles such scenarios. In spite of this increase, the GPU approaches still provided over an order of magnitude speedup over the CPU baseline. When looking at only the processing times (time spent by the GPU), note that the timings are consistent with those when data fits in main memory. Even when executing a query with close to 2.3 billion points and over 3,900 polygons, the GPU processing time with Bounded is *less than 5 seconds*. Since the counties data spans the whole USA, we chose $\epsilon = 1$ km for the

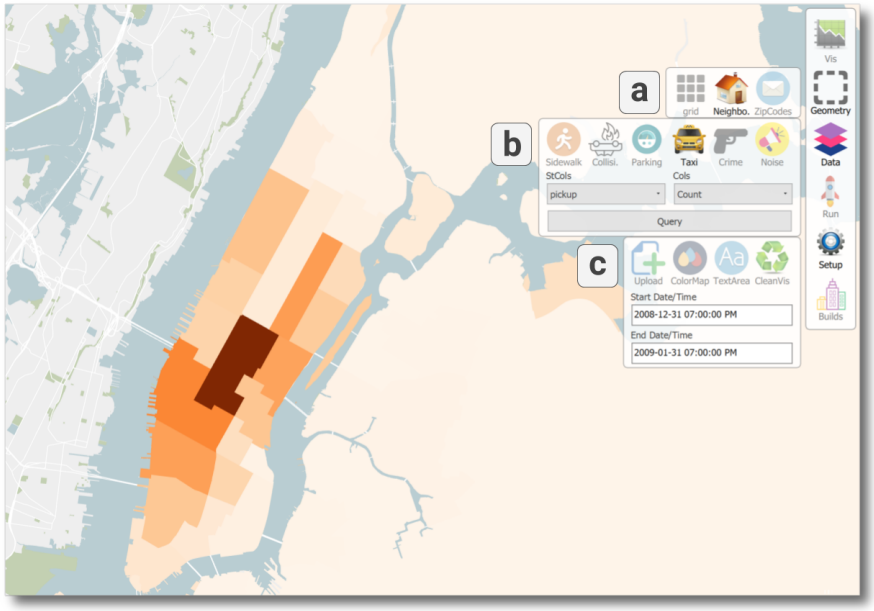


Figure 3.16 – The map view of Urbane. The density of NYC taxi data (b) is visualized over the neighborhood regions (a) for a chosen time range (c). The menu highlights this selection.

above experiments. Figure 3.15 shows the accuracy-time as well as accuracy-ϵ-bound trade-off for 1.8 billion points. The scatter plot visualizing the accuracy of the query is similar to the taxi experiments, with the points falling close to the diagonal.

3.8 Integrating Raster Join into Urbane

In this section, we provide some more details about Urbane [51], and describe how we integrated Raster Join into Urbane to speedup the visual exploration.

3.8.1 The Interface of Urbane

The visual interface of Urbane is comprised of two components: the *Map View* and the *Data Exploration View*.

Map View. This view is composed of a map rendering component (see Figure 3.16). The various menus and panels are overlaid on the map. Navigation and operations on map view such as panning, zooming, and rotating the view are accomplished through mouse interactions. The main menu (right side of map view in Figure 3.16) allows users to control all the functionalities of the system. This includes loading or deleting urban data sets as well as polygonal regions that define the different resolutions. Users can then choose the data set to be visualized along with the visualization resolution. For example, in Figure 3.16, the NYC taxi data is chosen

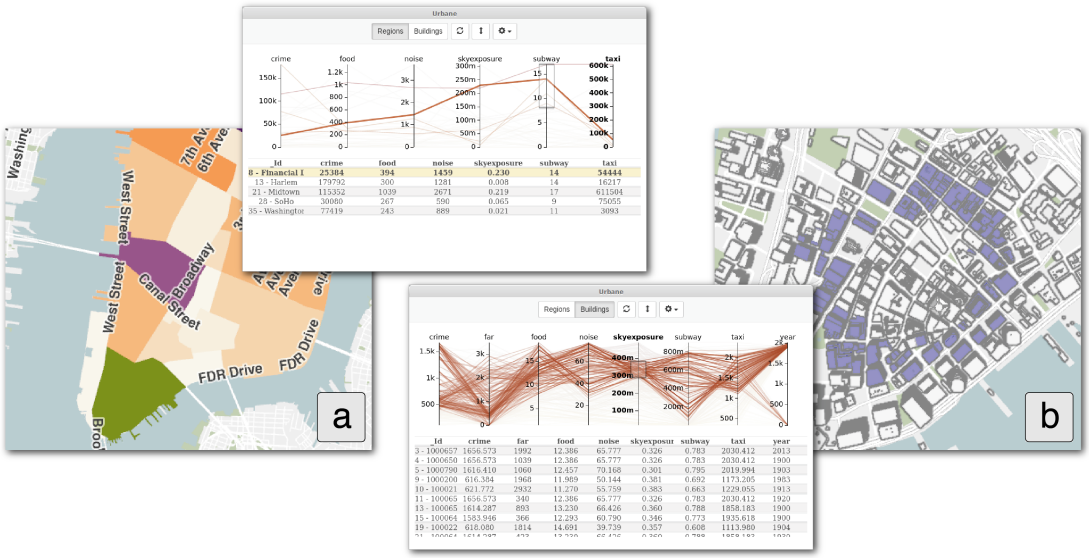


Figure 3.17 – Multi-resolution exploration. Neighborhoods having a high density of subway stations are highlighted, and Financial District is selected for further analysis (a). Exploring buildings in the selected region to identify opportunities for new development (b).

to be visualized and the aggregation is performed over the neighborhoods of NYC (chosen polygonal regions). The menu also allows users to toggle the data exploration view that enables a comparative exploration of the different data sets.

Data Exploration View. The main goal of the data exploration view is to support the analyses of urban data at two different resolution levels—*region* and *building*. This view consists of two components—a parallel coordinates chart (PCC) and a data table (see Figure 3.17). At the region level, the PCC allows users to visually analyze and compare multiple data sets across different polygonal regions (selected via the menu), and the data table shows the aggregate values for each of the regions. In the PCC, each data set (or dimension) is represented as a vertical axis, and each region is mapped to a polyline that traverses across all of the axes, crossing each axis at a position proportional to its aggregate value for that dimension. Users can also filter regions by brushing the desired range of values on individual axes of the PCC. This updates the map by highlighting all regions that satisfy the filter constraints (Figure 3.17(a)).

Users can also drill down into the building level by selecting a region of interest from the data table, and choosing the building option. At the building level, users can perform a similar exploration as above, i.e., visualize and analyze the different data sets, but in the context of each of the buildings within the selected region. This operation is illustrated in Figure 3.17(b). Here, the value for each building is computed by aggregating the data within a fixed radius of the building.

3.8.2 Integrating with Raster Join

Urbane generates spatial aggregation queries for two different operations: visualizing on the map, and visualizing on the PCC. At the region level, the polygonal data set corresponds to the different regions chosen by the user. At the building level, the polygons used in the join correspond to circles centered around the different buildings within the selected region. The accuracy (ϵ-bound) can be controlled by the user, with 20 meters being the default value for NYC. The Raster Join approach primarily takes as input a set of points and a collection of polygons, and computes the spatial aggregation as the output. Even though it also supports filtering the data over multiple attributes, it could still become inefficient to execute Raster Join over an entire data set due to the memory transfer overhead between the CPU and the GPU. Thus, to reduce this overhead, we store the different urban data sets in a 3D grid index of fixed size, where the dimensions correspond to the location (lat/long coordinates) and time. The extent of the grid in time is provided as a hint by the user, while in space it depends on the city that is being explored. To handle outlier points that lie outside the defined extent, we simply associate them with the closest grid cell. Based on the query parameters—the time range and the extent of the polygonal data sets, only data from the appropriate grid cells are transferred to the GPU for further processing. In practice, this approach significantly reduces the amount of data that is being transferred to the GPU. Other indexing methods could also be employed to efficiently perform this pre-filtering.

In a system like Urbane, spatial aggregation queries can be generated at high rates based on the user interactions. Bounded Raster Join provides efficient support for these queries. As we showed in the previous section, Bounded Raster Join can execute on-the-fly queries involving over 868 million points in only 1.1 second on a current generation laptop. Therefore, by integrating Raster Join into Urbane, we allow users to interactively explore, over space and time, several urban data sets at multiple resolutions. Finally, we note that Raster Join is not specific to Urbane or visualization systems. It is an efficient *operator* to perform spatial aggregation that has utility in a variety of applications in different fields and can be integrated into any system .

3.9 Limitations and Discussion

Worst-Case Scenario for the Accurate Approach. When the polygonal data set is very dense, every pixel of the FBO will fall at the boundary of some polygon and the accurate variant essentially becomes the baseline index-based approach. In fact, in this case, the accurate variant will take more time than the baseline, as it performs additional drawing (rendering) operations. This can also happen if the data is skewed such that all points fall close to the boundaries of the polygons.

Choice of Color Maps. We assume that continuous color maps are used for visualizations. In the case of categorical color maps, for values that fall around the boundary of two colors, even a minute error can completely change the color of the visualization.

Choosing Between the two Raster Variants. Setting a very small bound can result in the accurate variant becoming faster than the bounded variant of the raster join. This is because of the high number of renderings required to satisfy the input bound. We intend to add an estimate of the time required for the two variants, so that an optimizer can choose the best option based on the input query.

Performing Multiple Aggregates. Our current implementation performs only one aggregate per query. For multiple aggregates, multiple queries have to be issued. However, the implementation can be extended to support multiple aggregate functions by having multiple color attachments to the FBO. Similar to the multiple constraints scenario, this will increase the memory transfer time.

3.10 Chapter Summary

In this chapter, we propose efficient algorithms to evaluate spatial aggregation queries over arbitrary polygons, which is an essential operation in visual analytics systems. By efficiently making use of the graphics rendering pipeline and trading-off accuracy for performance, our approach achieves real-time responses to these queries and takes into account dynamic updates to query parameters without requiring any memory-intensive pre-computation. In addition, the OpenGL implementation makes the technique portable and easy to incorporate as an operator in existing database systems.

By showcasing the utility of computer graphics techniques in the context of spatial data processing, we believe this work opens new opportunities to make use of advanced graphics techniques for database research, especially in the context of the ever increasing spatial/spatio-temporal data. For example, spatial joins between 3D data sets could greatly benefit from the use of ray casting and collision detection approaches. These approaches could also be applied to perform more complex spatio-temporal joins.

4 Clipping Minimum Bounding Boxes for Efficient Spatial Data Exploration

The majority of spatial data processing techniques rely heavily on approximating groups of spatial objects by their minimum bounding box (MBB). As each MBB is compact to store (requiring only two multi-dimensional points) and intersection tests between MBBs are cheap to execute, these approximations are used predominantly to perform the (initial) filtering step of spatial data processing. However, fitting (groups of) spatial objects into a rough box often results in a very poor approximation of the underlying data. The resulting MBBs contain a lot of "dead space"—fragments of bounded area with no actual objects—that can significantly reduce the filtering efficacy.

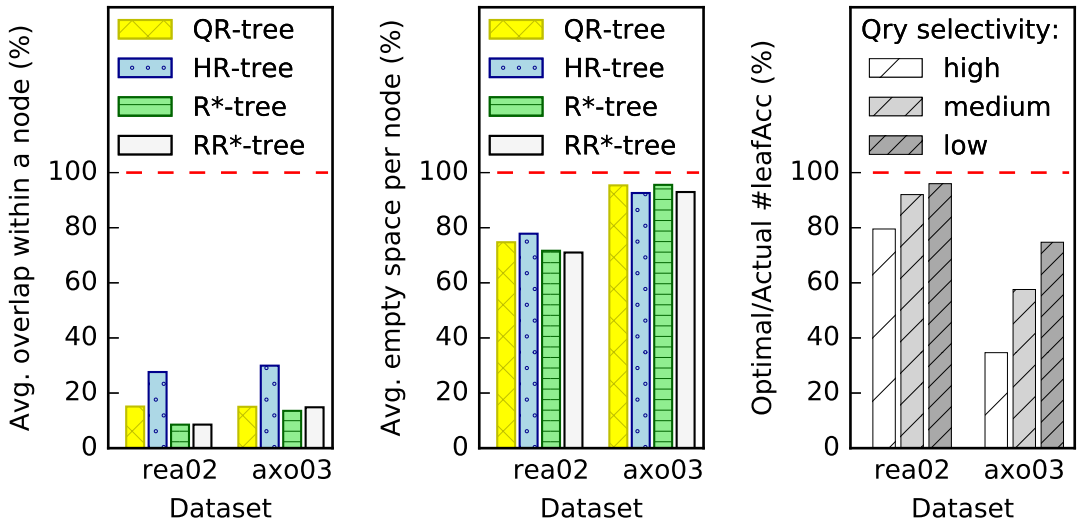
This chapter[1] introduces the general concept of a *clipped* bounding box (CBB) that addresses the principal disadvantage of MBBs, their poor approximation of spatial objects. Essentially, a CBB "clips away" dead space from the corners of a MBB by storing only a few auxiliary points. On four popular R-tree implementations (an ubiquitous application of MBBs), we demonstrate how auxiliary CBB points can be exploited to avoid many unnecessary recursions into dead space with minor modifications to the query algorithm. Extensive experiments show that clipped R-tree variants substantially reduce I/Os: clipping the state-of-the-art revised R*-tree, for instance, eliminates on average 19% of I/Os.

4.1 Introduction

Spatial data is growing at an alarming rate, prompting all major database system vendors to add spatial extensions that explicitly target spatial data analysis. From Oracle Spatial [91] and IBM Informix [74] to PostGIS [127] and HyPerSpace [120], these extensions rely heavily on the R-tree [66] spatial index. The fundamental component of the R-tree and all its variants, and in fact most spatial processing techniques, is the minimum bounding box (MBB). Consequently, even minor improvements to MBBs can have a broad impact.

[1] The material of this chapter has been the basis for the ICDE 2018 paper *Improving Spatial Data Processing by Clipping Minimum Bounding Boxes* [135].

Minimum Bounding Box. The MBB is the smallest axis-aligned rectangle that encloses its *d*-dimensional data. Being in a *conservative* class of approximations [26], it can represent any set of spatial objects (from simple points to complex volumetric shapes or other MBBs). Generally, a group of spatially-near objects are stored together (inside index nodes or buckets) and approximated by their MBB, offering three advantages: MBBs (i) are computed simply, with linear cost, (ii) are very compact to store, needing only two spatial points, and (iii) are very cheap to compare to each other for overlap/intersection. This is critical, as MBB intersection tests are the most dominant operation in spatial indexing: R-trees rely heavily on them during both the building and querying phases, while many other spatial tasks use them for filtering, e.g., traversing quadtrees [129] or performing spatial joins [28, 122].



(a) Overlap **(b) Dead space** **(c) I/O optimality**

Figure 4.1 – Performance of four R-tree variants.

Overlap, coverage, and "dead space". The quality of spatial data partitioning is classically measured by the *overlap* and *coverage* of the resultant MBBs. High overlap (i.e., large amount of area covered by multiple MBBs) decreases the filtering precision. This is particularly crucial for R-trees [23]: if a query rectangle intersects overlapping MBBs, the query must follow several paths in the tree. Reducing coverage (i.e., avoiding unnecessarily large MBB volume), however, is also important. Over-coverage increases the likelihood that a query rectangle will intersect a MBB, independent of the likelihood of intersecting the constituent objects. There has been extensive research on minimizing the overlap in R-trees, resulting in the proposal of several variants (R^+-tree [131], R*-tree [15], RR*-tree [17] to name a few).

In general, existing techniques minimize overlap quite well. In Figure 4.1a, we use two real-word data sets (described in Section 4.6.2), including a novel and challenging brain axon data set, to construct four popular disk-based R-tree variants (including the state-of-the-art, RR*-tree [17]) and measure the amount of overlap. Indeed, on both data sets and all four variants, just 8–30% of

the area of a node, averaged over all internal nodes, is overlapped by two or more of its children. However, Figure 4.1b is less encouraging. It shows what we call *dead space*, i.e., the percentage of the volume of a node that does not contain any objects. While one expects unnecessary coverage in higher dimensions due to the curse of dimensionality, we see a staggering amount, circa 74% and 94%, of dead space already in these 2*d* (`rea02`) and 3*d* (`axo03`) spatial data sets. This demonstrates the difficulty in bounding real objects such as road networks (`rea02`) and brain axons (`axo03`) with MBBs, even if they are easily separable.

Finally, Figure 4.1c illustrates that these hardly-overlapping but mostly empty MBBs indeed have a negative effect on query performance. We query the (state-of-the-art) RR*-tree [17] under three settings of query selectivity that return a couple (high selectivity), about ten (medium), or roughly one hundred (low) objects. The plot reports the fraction of leaf node accesses—the major I/O bottleneck—that actually contribute to the query result (i.e., contain at least one spatial object within the query range). High/medium selectivity queries, common in spatial joins [28, 122], are particularly affected; the query intersects just dead space in 21% (2*d*) and 64% (3*d*) of the accessed leaf nodes. For the other three R-tree variants (not shown), the results are even more discouraging.

Our clipping proposal. Certainly, we are not the first to observe the limitations of bounding boxes; various polygons [65, 82] and conics [41, 87, 157, 164] have been proposed and compared [26, 27]. However, these are limited by: (a) the complexity of their representation, (b) the complexity of their intersection tests, and/or (c) the lower bound on their dead space imposed by their convexity.

In contrast, we propose simple, non-convex polygons obtained by rectangularly clipping off the corners of MBBs (c.f., Figure 4.2 on the next page). Incidentally, the corners are the convergence points of the dimension-wise maxima/minima of the bounded objects and thus where much of the dead space is concentrated. Each clip requires only one *d*-dimensional point (and a *d*-bit flag) because the corner of the original MBB provides the opposing corner of the rectangular clipped area. Testing intersection with a clipped corner (we will show) is even cheaper than the preceding intersection test with the MBB. Finally, because the clipped corners are supplemental to the original MBBs, we can implement the proposal as a plugin-like addition for any R-tree variant: we store clip points in a small, auxiliary data structure, post-process MBBs at construction time, and minimally expand the query algorithm.

Clipping MBBs removes dead space and thereby achieves both classic objectives of spatial data partitioning: coverage is assuredly reduced by not representing dead space in the corners; overlap is potentially reduced by bounding the objects more tightly. Ideally we introduce very few clips that eliminate most dead space. We propose two means of generating clip points, based on the idea of Pareto optimality (i.e., skylines [22] in database literature), that trade-off complexity for pruning power. Our experiments with benchmark R-tree queries show that these proposals reduce leaf node accesses by 14% and 26%, while introducing a storage overhead of just 3.2% and 6.5%, respectively (averaged across seven data sets of 2–3 dimensions and four R-tree variants).

Moreover, for two classic spatial join strategies, the Index Nested Loop Join and the Synchronised Tree Traversal, we eliminate 46% and 18% of I/Os, respectively (averaged across four R-tree variants).

Contributions and outline. In this chapter, we propose low-overhead improvements to minimum bounding boxes (MBBs) to improve their ability to represent complex, real spatial objects. While Section 4.3 details related work and Section 4.8 concludes, our main contributions include:

- Introducing the general concept of *clipped bounding boxes* (CBBs) along with two particular instantiations of the concept (Section 4.4);
- Demonstrating that our CBBs can be plugged into any R-tree variant with a small auxiliary data structure and minor modifications to the construction, query, and update algorithms (Section 4.5);
- Experimentally showing that with three-fold fewer corners, our CBBs can prune more area than the convex hull, while providing average I/O savings of 29 %, 29 %, 27 %, and 19 % when plugged into the QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. In addition we are showing that our CBBs can save up to 53% of I/Os for the state-of-the-art spatial join strategies when plugged into R-tree variants (Section 4.6).

4.2 Background and Context

R-trees [66] are arguably the most popular spatial indexes. They are height-balanced (with all leaf nodes at the same level) hierarchical structures designed for organizing a set of d-dimensional rectilinear rectangles (or MBBs). The MBB approximations of data objects are stored in leaf nodes and intermediate (directory) nodes are built by grouping rectangles at the lower level (up to a maximum node capacity M). Rectangles at each level can be overlapping, covering each other, or completely disjoint. Since R-trees partition the actual data (into nodes), they obey the actual spatial object shapes and are particularly attractive for indexing spatially-extended (volumetric) objects (as opposed to space-partitioning indexes, e.g., Quadtree [129]).

In a disk-based setting, M is usually set to match the disk page size (e.g., 4 KB) or a small factor thereof and the minimum occupation of a node (thereafter denoted by m) is set to $\leq \lceil M/2 \rceil$. Upon insertion, if a node becomes overfull (i.e., already contains M objects), a dedicated node splitting algorithm is triggered that is responsible for distributing the objects into two (or more [23]) groups. Upon deletion, if a node contains less than m objects, a node merging algorithm is triggered. Both algorithms are crucial in maintaining nodes with both minimal coverage and overlap. As such, there has been a plethora of research resulting in R-tree variants that mainly differ in their strategy for distributing rectangles within and across nodes.

The originally proposed quadratic R-tree [66] uses solely the criterion of minimum coverage. The R*-tree [15] uses multiple criteria including minimizing dead space and margin in each node as well as the overlap between nodes. The state-of-the-art revised R*-tree [17] employs further more sophisticated techniques (e.g., switching between perimeter- and volume-based optimizations) to drive the insertion strategy. The Hilbert R-tree [85] imposes a linear ordering on the rectangles (by transforming them to one-dimensional, locality preserving Hilbert values) and uses this ordering to distribute rectangles to nearby nodes. However, the latter is mainly for static scenarios (i.e., requires a predefined data space). More details are in Section 4.3. Nonetheless, despite these differences, the physical layout (c.f., Figure 4.4a) and general construction/query algorithms are the same for all variants.

Nowadays, as scientists in various disciplines no longer rely solely on studying subjects of interest in their laboratory or in nature, but complement their understanding of the phenomena by building spatial models of increasing details, spatial data sets are ever-growing. Furthermore, scientific and other similar applications give rise to spatial data that does not conform well to rectangular bounding. As we have shown above, one example of such challenging data is $3d$ brain models built by neuroscientists. Since neurons tend to be elongated and non axis-aligned, it is inherently difficult to fit them into rough rectangular boxes.

While in the above we exemplify the motivating problem using R-trees, the general problem is that of the poor representation of a collection of spatial objects using minimum bounding boxes. Consequently, it is present in many MBB-based filtering techniques in spatial data processing (e.g., spatial join [28, 122]).

4.3 Related Work

The R-tree family. The R-tree [66] is a disk-based multi-dimensional index structure consisting of a hierarchy of minimum bounding boxes (MBBs) which recursively enclose data objects. Indexing and query processing with R-trees in spatial databases have received a lot of research attention [103] and several variants of the original approach have been proposed. To increase robustness against different data distributions, the R*-tree [15] incorporates an improved node split algorithm and the removal and reinsertion of spatial objects of an overflowing node. It employs multiple optimisation criteria at every split, attempting to minimise the dead space and the margin in each node as well as the overlap between nodes. The RR*-tree [17] introduces more adaptive optimisation strategies in order to further reduce I/O costs and enhance search performance.

Bulk loading can optimise the partitioning of data in the R-tree during initial construction. The HR-Tree [85] uses the Hilbert space filling curve to identify spatially close objects, while STR [95] recursively sorts the objects along each dimension for spatial proximity. To better handle extreme data, the PR-Tree [12] groups all objects with extreme coordinates in the same dimension in the same node.

All the above and many other access methods [54] operate on MBBs and thus our proposed clipping techniques can be applied orthogonally to boost their query performance.

Space-oriented partitioning. Instead of grouping objects hierarchically based on their location, another family of spatial indexing methods splits space using hyperplanes into a set of disjoint partitions that are stored flatly [6] or in a hierarchical structure [80, 129]. To achieve good storage utilization, the hB-tree [101] does not require that nodes are split by hyperplanes; instead, it allows them to represent hyperrectangular regions with hyperrectangular holes. Note that, similarly to CBBs, the hole can be located at the corner of the node. However, in such cases, the corner region is not a dead area, but is in fact *guaranteed to contain data objects*. By definition, space-oriented partitions do not minimally bound the enclosed data objects and therefore contain dead space.

Bounding objects. Instead of using minimum bounding boxes to represent a collection of objects, other geometries have been proposed in the past. The Sphere-tree [157] is a hierarchical structure similar to the R-tree with the exception that it uses minimum bounding spheres. The Sphere-tree requires less storage space than the R-tree but the computation of the minimum bounding sphere is more expensive [163]. The SS-tree [164] indexes multidimensional points and employs bounding spheres having as center the centroid of the underlying points for the region shape. The SR-tree [87] integrates both bounding spheres and bounding boxes to reduce the volume that an index node occupies and increase search efficiency. However, the SR-tree has lower fanout (its nodes contain both a sphere and a box) and higher creation cost compared to the SS-tree. The eR-tree [41] is a variant of the R-tree that employs minimum volume covering ellipsoids. Ellipsoids cover less dead space on sparse clustered data but their advantage is suppressed on dense data. Moreover, they cover more dead space than polygonal alternatives [26, 27]. The P-tree [82], another generalization of the R-Tree, uses a hierarchy of polygon containers that are mapped into rectangles of a higher dimension. The use of additional hyperplanes results in a better approximation, but it increases the size of the entries (i.e., reduces the fanout of interior nodes). The Cell Tree [65] is a hierarchy of nested convex polyhedra that are subspaces of a binary space partitioning and aims to handle objects of arbitrary shape. In contrast, our work is not proposing an alternative geometry, as MBBs have been tremendously successful in practice. Instead, we extend MBBs using clip points that adhere to the same basic, rectilinear concept.

Recently, [161] tackled the same challenge, "over-coverage" of MBBs, focusing solely on streaming observational data where a group of (successive) points can be represented as line segments. Our proposal is more general, supports points (lines/planes) as well as volumetric spatial objects, and does not imply any restrictions on the data (e.g., value-continuity of observations).

4.4 Eliminating Dead Space in MBBs

Figure 4.2 gives an example that runs over the next two sections. It depicts a set $O = \{o_1, \ldots, o_5\}$ of five (gray) spatial objects and the (black) box that minimally bounds them. It is clear that

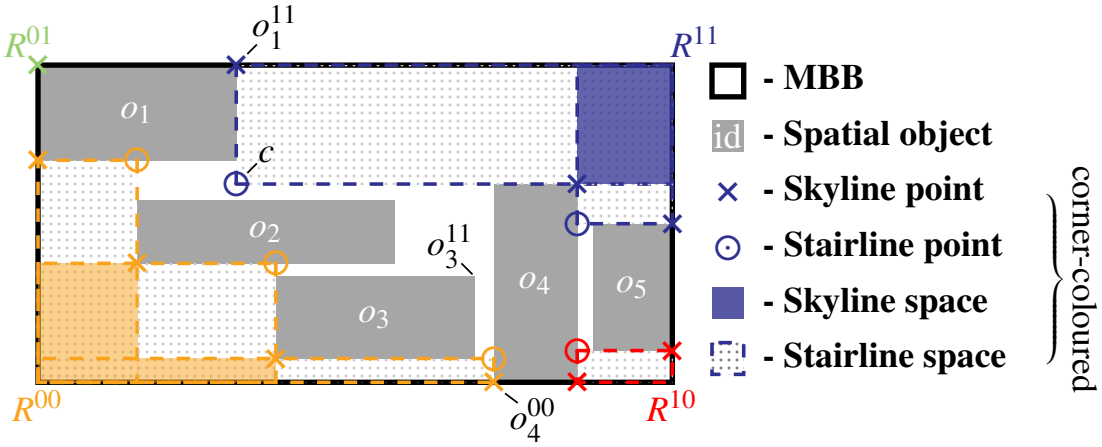


Figure 4.2 – Concepts related to *clipped bounding boxes*.

most of the space inside this MBB is not covered by an object, but is instead "dead space". We show in this section how to eliminate a lot of that dead space using only a few well-chosen points and bitmasks. These points "clip away" corners of the MBB, producing what we call a clipped bounding box. Section 4.4.1 introduces the general concept and Sections 4.4.2–4.4.3 define two specific implementations.

4.4.1 The Clipped Bounding Box (CBB)

High-level concept. Intuitively, a lot of the dead space in an MBB is tucked into the corners, where the most extreme values of all the objects converge. Incidentally, the rectangular area A adjacent to a corner is also the cheapest to represent: A, like any box, is defined by two points, one of which is the corner point; so, a single d-dimensional point and a d-bit flag to identify the targeted corner are sufficient to fully characterise A. Our proposed *clipped bounding box* (CBB) augments an MBB with pairs of additional points and corner flags to *clip away* excess area near the corners of the MBB. The reduction in dead space can be quite profound. In Figure 4.2 for example, one can store the point c with the bitmask $b = 11$ to represent the rectangle $\langle c, R^{11} \rangle$, which contains *no* objects. If this region was not part of the MBB, one would have an equally correct, but more representative, bounding object. Moreover, the cost of intersecting a query region Q with this clipped rectangle is low: we need only compare Q to the MBB (which, anyway, would be necessary), and then to point c.

In general, there are many choices for c and we need not select only one. Choosing the right points is, unsurprisingly, non-trivial and thus the subject of Sections 4.4.2–4.4.3. First, however, we formalise this high-level intuition by introducing notation and definitions.

Notation. An MBB in d dimensions is a hyperrectangle, which we denote by $R = \langle l, u \rangle$. The two points defining R, l and u, represent the minimum and maximum extent of R, respectively. We express coordinates of a point $p = (p[1], \ldots, p[d])$ and the bits of a bitmask $b = \langle b[1] \ldots b[d] \rangle$ in

array notation. The *minimum bounding box* (MBB) of a set of m objects $O = \{o_1, \ldots, o_m\}$ is the smallest possible rectilinear box $R = \langle l, u \rangle$ that contains all objects in O; i.e., for each dimension i, $l[i] = \min_{o_j \in O} o_j[i]$ and $u[i] = \max_{o_j \in O} o_j[i]$. We frequently refer to specific corners of bounding boxes (hyperrectangles). For hyperrectangle R a unique superscripted bitmask b specifies the corner of interest: a set bit $b[i]$ indicates that the corner maximises the i'th dimension; i.e.,

$$R^b[i] = \begin{cases} u[i], & \text{if } b[i] = 1 \\ l[i], & \text{if } b[i] = 0. \end{cases}$$

The four corners of R, the MBB in Figure 4.2, are labeled with this notation. For example, the top-left corner is R^{01} and the top-right corner is R^{11}. Similarly, the top-right corner of object o_1 is denoted o_1^{11} and the bottom-left corner of object o_4 is denoted o_4^{00}.

Formalisation. In general, an MBB is an imperfect approximation of a set of objects O, which a CBB improves. The extra, empty space that an MBB uses to bound O we call *dead space* and define as the part of R not occupied by any o_i of O (Definition 1):

Definition 1 (Dead space). Let R be the MBB of objects $O = \{o_1, \ldots, o_m\}$. The *dead space* of R, denoted $\dagger(R)$, is:

$$\dagger(R) = \{p \in R : \forall o_i \in O, p \notin o_i\}.$$

A CBB augments an MBB with extra points that we call *clip points* (Definition 2). A clip point is a pair consisting of a point $p \in R$ and an orientation mask b and has the property that no object $o_i \in O$ occupies the space between p and its relevant corner R^b. (Observe that the space between p and R^b is exactly the MBB of the two points $\{p, R^b\}$.)

Definition 2 (Clip point). Let R be the MBB of objects $O = \{o_1 \ldots, o_m\}$ and b be a bitmask of length d. We say that $\langle p \in R, b \rangle$ is a *clip point* iff the area between p and R^b is entirely dead space; i.e., if R' denotes the MBB of $\{p, R^b\}$, then p is a clip point iff:

$$\forall q \in R', \forall o_i \in O, q \notin o_i.$$

For example, $\langle c, 11 \rangle$ is a clip point in Figure 4.2, because the area enclosed by the dashed blue lines is empty. On the other hand, $\left\langle o_4^{00}, 11 \right\rangle$ is *not* a clip point, because it would clip away objects o_4 and o_5: the area between o_4^{00} and the top-right corner, R^{11}, is not entirely dead space.

We denote the volume that clip point $\langle p, b \rangle$ clips away by $\mathrm{Vol}_R(\langle p, b \rangle)$. In general, we clip away several parts of an MBB using a set of $\langle p_i, b_i \rangle$ pairs, but take care not to double-count regions clipped away by multiple clip points. That is to say, given a set of clip points $P = \{\langle p_i, b_i \rangle\}$, $\mathrm{Vol}_R(P) = \bigcup_{\langle p_i, b_i \rangle \in P} \mathrm{Vol}_R(\langle p_i, b_i \rangle)$.

This leads us to our core concept, a *clipped bounding box* (CBB), that we informally introduced at the beginning of this subsection and that we formally define in Definition 3:

Definition 3 (Clipped Bounding Box (CBB)). Given a set of objects O, a *clipped bounding box* is a pair $\langle R, P \rangle$, where R is the MBB of O and P is a set of clip points in R.

Naturally, a CBB, $\langle R, P \rangle$, is a better approximation of O than another CBB, $\langle R, P' \rangle$, if it retains less volume; i.e., $\text{Vol}_R(P) > \text{Vol}_R(P')$. While a clip point introduces very little overhead, a large set of clip points is cumbersome. Thus, we only want to select $\leq k$ of the clip points that we propose in the following subsections, while still maximising $\text{Vol}_R(P)$.

4.4.2 Object-situated Clip Points

High-level concept. Given an MBB R, we can obtain good clip points quickly by taking them from the objects O bounded by R. Consider Figure 4.2 again; (a discretisation of) the set of possible clip points is depicted by small, gray dots. For the most part, the best clip points (i.e., the dots farthest from their respective corners) lie on the outer surface of some object o_i. This is intuitive: the dead space arises specifically because the MBB corner is too far from the objects, so clipping R with a rectangle that borders an object o_i will naturally improve the approximation of O.

With respect to a corner R^b, we do not consider all (infinitely many) points on the surface of each object $o_i \in O$, but rather just its closest corner, o_i^b. This is most likely to be a valid clip point. In fact, if $\left\langle o_i^b, R^b \right\rangle$ is not a clip point, then *no* point in o_i can be a clip point with respect to R^b. Considering, for example, corner R^{11} in Figure 4.2, we see that $\left\langle o_3^{11}, R^{11} \right\rangle$ is not a clip point (it would clip away part of o_4 and o_5), so the entirety of o_3 would also clip away part of o_4 and o_5 and therefore similarly not be clip points.

We also do not necessarily consider all objects $o_i \in O$, because we recognise that, for corner R^b, the subset of $\{\left\langle o_i^b, R^b \right\rangle\}$ that are clip points is precisely the well-studied concept of a skyline [22], computed over $\{o_i^b\}$. Thus, the idea for our object-based CBB is to compute for each bitmask b the skyline of the M object corners $\{o_i^b\}$ and pick from those the few points that clip away the most dead area. Thus the clip points are all actually represented in the set of underlying objects, O. To formalise this intuitive description, we will state the definition of a skyline as it relates to this context.

Oriented skyline of objects. The skyline [22] is based on the concept of *dominance*, which is highly related to our notion of clip points. A point p dominates a distinct point q with respect to b if it is at least as close to R^b as q on each dimension independently. More precisely, let $b_{p \odot q}$ denote a bitmask with bit i set iff $p[i] \odot q[i]$. Then we have Definition 4:

Definition 4 (Dominance). Given points p and q, and bitmask b, p dominates q w.r.t. b, denoted $p \prec_b q$, iff:

$$\left(b_{p \leq q} \;\&\; {\sim}b = b_{p \leq q}\right) \wedge \left(b_{q \leq p} \;\&\; b = b_{q \leq p}\right) \wedge b_{p \neq q} \neq 0.$$

For example, given $b = 00$ in Figure 4.2, $o_4^{00} \prec_b o_5^{00}$ because it is closer to R^{00} in both the x and the y direction. In the context of MBBs, one could express dominance equivalently by letting R' denote the MBB of $\{q, R^b\}$ and stating that $p \prec_b q$ iff $p \in R'$. Using the same example, observe that the point o_4^{00} indeed lies in the MBB created by the point it dominates, o_5^{00}, and R^{00}. The *oriented skyline* of a set of points P, given orientation mask b, is simply the subset of points $p \in P$ not dominated by any other point $q \in P$:

Definition 5 (Oriented skyline). Given point set P and orientation mask b, the *skyline* $\mathcal{S}_b(P)$ is:

$$\mathcal{S}_b(P) = \{p \in P : \nexists q \in P, q \prec_b p\}.$$

Considering an MBB R and a set of object corner points $P = \{o_i^b\}$, a pair $\langle p \in P, b\rangle$ is clearly a clip point iff $p \in \mathcal{S}_b(P)$. If a point $p \notin \mathcal{S}_b(P)$ is dominated by some other point $q \in P$, then q lies in the MBB of $\{p, R^b\}$ which implies that $\langle p, b\rangle$ would clip away the object from which q was derived. On the other hand, if $p \in \mathcal{S}_b(P)$, then no other point $q \in P$ lies in the MBB R' of $\{p, R^b\}$. Since P contains the closest point to R^b from every object $o_i \in O$, the entirety of $\left\langle p, R^b\right\rangle$ must contain dead space. Thus, the clip points in $\{\left\langle o_i^b, b\right\rangle\}$ are in one-to-one correspondence with $\mathcal{S}_b(\{o_i^b\})$.

We see this in Figure 4.2. Considering corner $b = 00$, for example, we obtain a skyline of $\{o_1^{00}, o_2^{00}, o_3^{00}, o_4^{00}\}$. Point o_5^{00} is dominated by both o_3^{00} and o_4^{00}; meanwhile the clip point $\left\langle o_5^{00}, b\right\rangle$ would clip away part of o_3 and o_4.

4.4.3 Point-spliced Clip Points

High-level concept. We can find more aggressive clip points by *splicing* the skyline points proposed in Section 4.4.2. Recall the possible clip points in Figure 4.2. Skyline point o_4^{11} clips away a lot of dead space relative to R^{11}, but the point c that combines the y-coordinate of o_4^{11} with the x-coordinate of o_1^{11} clips away significantly more dead space. In fact, c clips away the most dead space (of those that could form valid clip points). c is not an arbitrary point in $\mathbb{R}^2$, but a combination of the coordinates of o_1^{11} and o_4^{11}: this *splicing* provides a generative mechanism of strong clip points that may not lie on any object at all, and comprises our second instantiation of CBBs. It clips away much more dead space than our first CBB proposal, but requires an expensive extra processing step.

Stairline points. We define *stairline points*, which, intuitively, are the points "between" skyline points, farthest from a corner R^b of the MBB. To find them, we introduce the *splice point* concept, which mixes the coordinates of source points p, q (thereby still being adjacent to child MBBs):

Definition 6 (Splice point). Given two points p and q with MBB R, their *splice point* with respect to b is $\Cat_b(p, q) \equiv R^b$.

Stated alternatively, $\Cat_b(p, q)[i]$ has value $\max(p[i], q[i])$ if $b[i]$ is set; otherwise, it takes the value $\min(p[i], q[i])$. For example, c in Figure 4.2 is equal to $\Cat_{00}(o_1^{11}, o_4^{11})$, i.e., takes the smallest x and y values from its source points o_1^{11} and o_4^{11}, as the bitmask $b = 00$ specifies to minimise both dimensions.

We must be careful when splicing points to not clip away occupied area, i.e., to ensure the generated clip points are "valid." Producing valid clip points in $2d$ is straight-forward: we can *totally order* all skyline points by x and then splice each consecutive pair. However, it is non-trivial in higher dimensions to efficiently extract all sets of neighbouring points.

Thus, we propose an unfortunately-cubic algorithm that is still practically reasonable given the small input sets ($< M$). We generate splice points using bitmask $\sim b$ from every pair of skyline points, some of which are invalid; to ascertain validity, we check that no other skyline point (and thus child MBB) is in the space that this splice point would clip away.

Definition 7 (Oriented stairline). Given a set of points P and a bitmask b, the *oriented stairline* is the subset of splice points, $\{\Cat_{\sim b}(p \in P, q \in P)\}$, $p \neq q$, that are clip points w.r.t. b. A point in the stairline is called a *stairline point*.

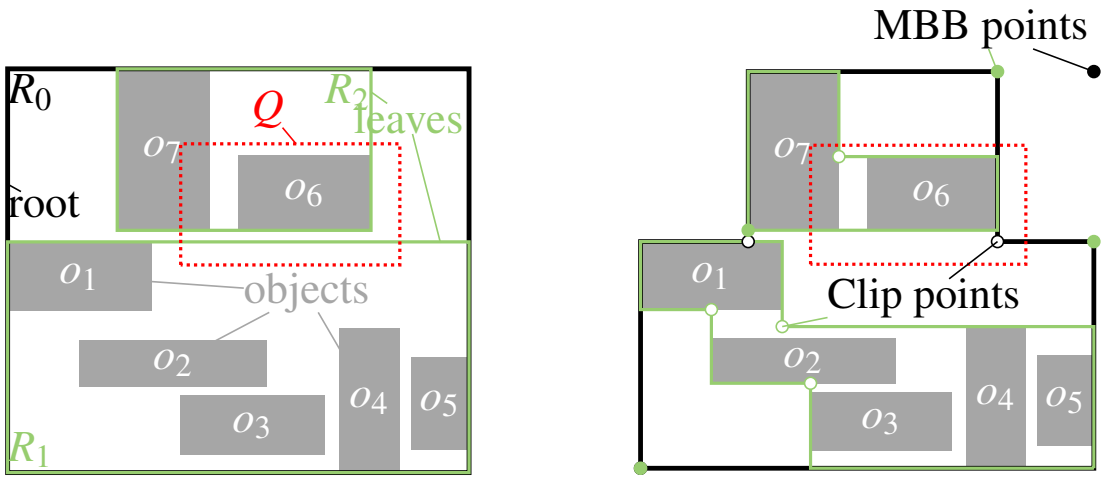
Stairline points necessarily clip away more dead space than the skyline points p, q from which they are spliced, because they take coordinates from both p and q such that they are farthest from the corner R^b.

4.5 CBB-based R-trees

Section 4.4 introduced the concept of clipped bounding boxes (CBBs) and two approaches to defining clip points for them. Here we describe how to integrate them into arbitrary R-tree variants. Recall that the R-tree variants vary in how they determine the contents of their nodes, but not in their general layout; thus, our extensions here apply to any variant.

4.5.1 Layout and Structure of Clipped R-trees

Figure 4.3 extends the example of Figure 4.2, contrasting a classic MBB-based R-tree (a) with a clipped one (b). In Figure 4.3a, the seven spatial objects (o_1–o_7) are indexed using the traditional R-tree with $M = 5$ and $m = 2$, which results in a two-level hierarchy of MBBs. The root node (in black) corresponds to R_0, which minimally bounds the two leaf nodes (in green). The



(a) A two-level R-tree **(b) A clipped R-tree**

Figure 4.3 – An example of an R-tree before (a) and after (b) clipping, given 7 objects, o_1–o_7 and a range query, Q.

R-tree distributes the objects into nodes well, as the leaves have zero overlap. The range query, Q, intersects two spatial objects: it fully covers o_6 and it partially intersects o_7. Nevertheless, because Q intersects all MBBs in the R-tree, all three nodes must be scanned, which unfortunately includes the node of R_1, inside which Q only overlaps dead space.

The clipped R-tree, for contrast, is shown in Figure 4.3b. Each of the MBBs (with corner points depicted by solid circles) from Figure 4.3a is independently augmented with clip points (depicted by hollow circles). Six clip points are introduced: two in R_0 (the black ones), three in R_1, and one in R_2. While Q still intersects R_0 and R_2 as they contain objects that are within Q's range, the excess leaf-node scan of R_1 is averted.

Figure 4.4a illustrates a typical R-tree data structure. The directory nodes (in this case just the root) contain an array with an MBB (R_0) followed by a list of between m and M children. For each child, its MBB and a pointer to the child node are given. The leaves contain an array with an MBB and a list of pointers to actual objects. To clip the R-tree, *we retain this original structure exactly* and augment it with the structure in Figure 4.4b that contains the clip points. A directory table is indexed by the ids of the R-tree nodes: entry 1 corresponds to R_1. It contains a length, since we adaptively determine the variable number of clip points per node, and a pointer to an array. Inside the array, the bitmask of the clip point is given first and the actual coordinates of the point are given next. The clip points are ordered by the volume that they clip away in order to detect non-intersection as quickly as possible.

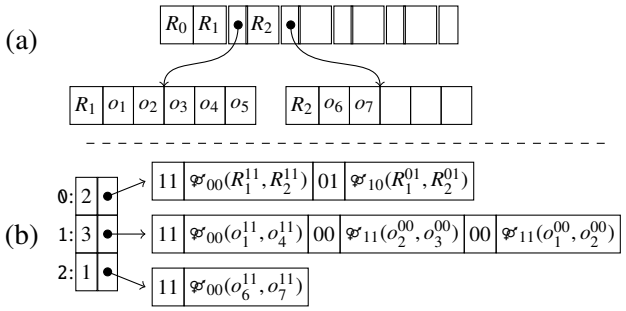


Figure 4.4 – The physical layout of the R-tree from Figure 4.3a (a) and the auxiliary structure (b) of clip points introduced in Figure 4.3b. The auxiliary table is indexed by MBB id and gives the number of and pointer to the clip points for that MBB. The bitmask of each (in this case $2d$) clip point is followed by the two coordinate values.

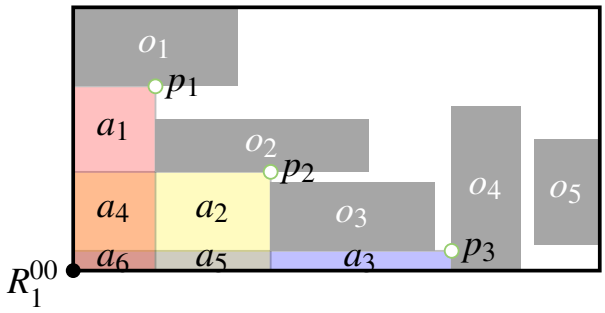
4.5.2 Constructing Clipped Bounding Boxes

In Section 4.4, we described two types of clip points, giving rise to two different clipped bounding boxes. Here, we describe how *to compute* those CBBs. In particular, we focus on the challenging problem of selecting k clip points from a set of choices that grows with M. We discuss this k-selection problem first, then turn to the high-level clipping algorithm.

Selecting k clip points. Determining the optimal selection of k clip points would incur exponential cost, because there is no known algorithm for $\geq 3d$ better than enumerating all possible size-k subsets [25]. However, we can avoid this cost with three reasonable simplifying assumptions: (1) each corner of a CBB is independent; (2) the point that independently clips away the most volume is among the k choices; and (3) if multiple points clip away area in the same corner, the overlap of that area is small and/or covered by the point in assumption (2).

We make assumption (1) because if clip points p and q arise from different corners, they can only clip away area that overlaps each other if there is substantial dead space. In that case, optimality is not crucial: greedily selecting k clip points will prune a lot of dead area, anyway. The simplification permits linearly combining the solutions from the 2^d corners, rather than computing cross-corner overlap.

Figure 4.5 illustrates the intuition behind assumptions (2) and (3). Three potential clip points, p_1–p_3, for corner R^{00} are shown, along with the area that each would clip away, area(p_i). If we pick multiple clip points for this corner, assumption (2), accurately in this case, asserts that p_2 would be in the optimal solution. Assumption (3) asserts that the overlap in clipped area among the multiple points will be contained in $\{a_2, a_4, a_5, a_6\}$ or very small. More specifically we assume that the point clipping away the largest volume (in this case p_2) will be selected *and* will clip away the overlap for all chosen points. Thus we assume that for another p_i, it will contribute area(p_i) – area($p_i \cap p_2$) to the overall union. Whereas exact computation would require invoking the exponential-cost inclusion-exclusion principle, we simply add these scores together to approximate the clipped area.



$$\text{area}(p_2) = \{a_2, a_4, a_5, a_6\}$$
$$\text{area}(p_1) = \{a_1, a_4, a_6\}$$
$$\text{area}(p_3) = \{a_3, a_5, a_6\}$$

$$\text{score}(p_2) = \text{area}(p_2)$$
$$\text{score}(p_1) = \text{area}(p_1) - \{a_4, a_6\}$$
$$\text{score}(p_3) = \text{area}(p_3) - \{a_5, a_6\}$$

Figure 4.5 – Demonstration of the overlap approximation. The combined score of $\{p_1, p_2, p_3\}$ overcounts the overlap of $\{p_1, p_2, p_3\}$ and undercounts the overlap of $\{p_1, p_3\}$, but these often correspond to the same area, a_6.

Algorithm 7: Clip: (node N, k, τ) $\longrightarrow$ set of clip points C

```
L ← empty list of clip points
for each bitmask b ∈ 0 ... (2^d − 1) do
    P ← S_b({o_i^b : o_i ∈ N.children})
    if using stairline points then
        for each s_i, s_j ∈ P do
            if ∀s_k ∈ P, ⋈_{~b}(s_i, s_j) ⊀_b s_k then
                P' ← P' ⋃ {⋈_{~b}(s_i, s_j)}
            end if
        end for
        P ← P'
    end if
    assign scores ∀p ∈ P as in Figure 4.5
    for each p ∈ P, with p.score > τ × area(N) do
        L.append(⟨p, b⟩)
    end for
end for
return the min(k, |L|) clip points in L with highest score
```

In this example, our approach produces the exact score. This occurs because p_1 and p_3 lie on opposite sides of p_2, which commonly happens: the point clipping away the most area is likely to lie on the diagonal and the next best choices (for small k) are likely to be on either side. Even when this intuition fails, the error of approximation is bounded by the intersection of the smaller rectangles, which itself is small.

As a last optimisation, we elect to only store clip points that clip away an additional $\geq \tau$ % of the volume; otherwise, they increase our storage cost without having a high likelihood of containing a query rectangle. Thus, we could end up with fewer than the intended number of k clip points if they do not prune much dead space, anyway.

Algorithm 8: Intersection Test: $(R, C, Q, \text{selector}) \longrightarrow$ `bool`

```
1: if Q ∩ R = ∅ then
2:     return FALSE
3: end if
4: for each c ∈ C do
5:     if Q^(selector⊕c.mask) ≺_c.mask c.coord then
6:         return FALSE
7:     end if
8: end for
9: return TRUE
```

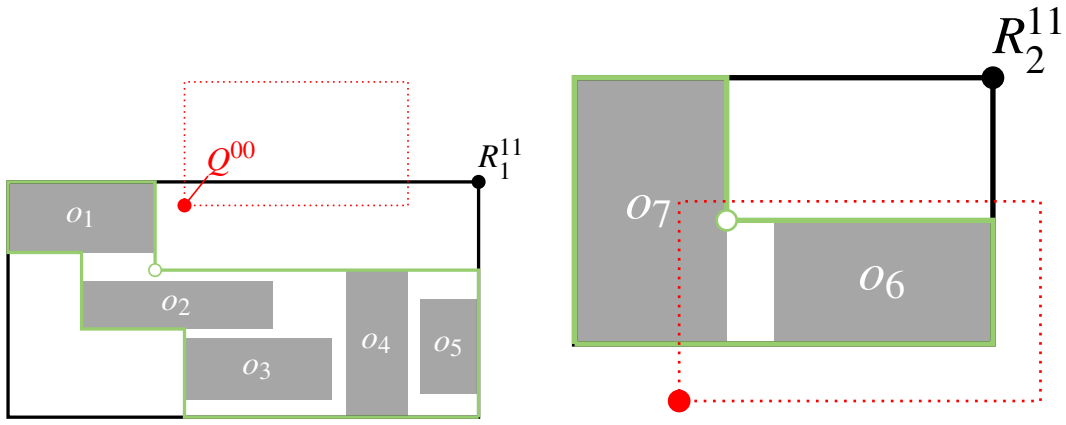
The clipping algorithm. To construct a clipped R-tree, we apply Algorithm 7 to every tree node and store each result with an entry in the auxiliary structure (Figure 4.4b). We iterate over the corners of the MBB independently (Line 2 and assumption 1), computing for each the skyline of the object corners (Line 3). We optionally splice the skyline points (Lines 4–10) to clip away more area. Then, we determine which clip point clips away the most area and assign approximate scores to the rest (Line 12 and assumptions 2 and 3). We finish processing the corner by keeping only the clip points passing the τ threshold (Lines 13–14). After iterating every corner, we sort the clip points by score and return the k highest-scoring.

4.5.3 Querying a Clipped Bounding Box

Range queries. Since CBBs use the corners of their MBBs, intersection can be done very efficiently, as shown in Algorithm 8. Lines 3–5 transform a typical MBB intersection test into a clipping-enabled one. Given clip points C, we simply check dominance (Definition 4) between the corner of the query rectangle $Q^b \in Q$ obtained by $Q^{\sim \text{c.mask}}$ and each clip point $c \in C$. (For queries, *selector* is fixed to $2^d - 1$: the *xor* expression ($\oplus$) is equivalent to negating *c.mask*. Its purpose will be clarified in Section 4.5.4.) Intuitively, it is the least "competitive" query corner that can dominate a given clip point. If any clip point is dominated by Q, then the CBB and Q are disjoint.

We query each leaf node of our running example in Figure 4.6. In Figure 4.6a, we compare Q to the first clip point, which is paired to corner R_1^{11}. (The clip points, recall, are sorted by the volume-based score.) Because Q^{00} dominates the clip point, relative to R_1^{11}, we know that the part of Q inside the MBB lays inside dead space and avoid scanning R_1. On the other hand, in Figure 4.6b, Q^{00} *does not* dominate the (single) clip point; so, we can conclude that Q intersects the CBB of R_2.

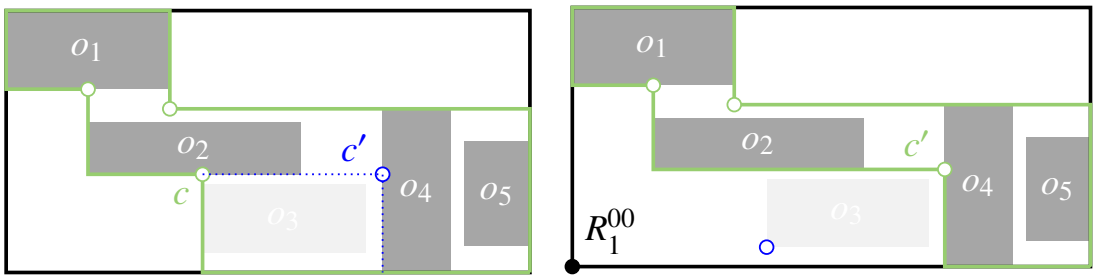
Spatial join. R-tree indexes are also leveraged for performing spatial joins. When only one data set is indexed, the spatial join can be evaluated by probing the index for each object of the other data set (Index Nested Loop Join evaluation). In this case, we are essentially performing range queries (one per object), therefore clip points are exploited as described above.



(a) Querying the bottom node.

(b) Querying the top node.

Figure 4.6 – Using *dominance* to test CBB intersections.



(a) Deleting object o_3.

(b) Inserting object o_3.

Figure 4.7 – Clip points before (green) and after updates (blue). Deleting o_3 creates a better clip point c', but c is still valid. On insertion, the blue corner of o_3 dominates c' with respect to the solid, black point, indicating that o_3 invalidates c'.

When both data sets are indexed, the join can be evaluated using the Synchronised Tree Traversal (STT) strategy. This strategy recursively restricts the search space only to the intersection of the MBBs of the corresponding sub-trees [28]. To benefit from clip points, we recursively restrict the search space to the intersection of the CBBs of the corresponding sub-trees and apply dominance tests to check whether a child CBB falls within it (comparison of the CBB of a child with the intersection of the CBBs of the parent level).

4.5.4 Updating a Clipped Bounding Box

An update to a memory-resident CBB is cheap, as a disk write to persist the changes in the underlying data is always coincident. Still, many unnecessary updates can be avoided.

First, we observe that any update that affects x MBBs can affect at most $x + 1$ CBBs. Unlike with the boundaries of MBBs, the changes to clip points do not need to propagate up the tree, because

each MBB is clipped independently. If the last MBB change occurs at level l of the tree, then the next shallower level, $l + 1$, is the last level at which clip points may change, as the clip points are based on the MBBs of the level below. Moreover, a split/merge that does not change an MBB also does not change the corresponding CBBs.

In general, if the MBB of node n changes, we recompute the clip points of n, because our thresholding with τ and our top-k ordering and selection are both distorted by the change: one must re-examine all candidate clip points for at least one corner, anyway. Below, we discuss how to avoid the additional $x + 1$'st CBB change for deletions and insertions. Modifications are handled by deleting and then re-inserting the object.

Deletions. Deletions are the easiest case, because they *generate new dead space*; thus, we can handle them "lazily". Figure 4.7a illustrates the deletion of object o_3 from our running example, which affects the CBB, but not the MBB, of the bottom node. Prior to the deletion, clip point c is in C. After the deletion, we could replace c with c', which prunes the new dead space. However, if we continue to use clip point c instead of c', we obtain accurate query results with the same pruning rate as prior to the deletion. Moreover, a subsequent insertion (as shown next), of either a new object or a reinsertion of o_3 (in the process of modifying it), only needs to be handled if it intersects the dead space pruned by the previous point c. Therefore, on deletions we never need to change a CBB if the MBB is unaffected.

Insertions. Insertions *remove dead space*, potentially invalidating our clip points; so, we handle them "eagerly". For each insertion, we check whether it invalidates one of our top-k clip points. If so, we recompute the CBB. If not, we are certain that our top-k selection is still correct, because the score of every other candidate clip point is either lowered or unaffected by the insertion. Figure 4.7b illustrates the insertion of o_3 (had it not been there before). Observe that c' prunes away space occupied by o_3, yielding incorrect query results. This can be detected by testing whether the blue corner 00 of o_3 dominates c' with respect to the black corner R_1^{00} of the MBB.

The validity test is identical to our query in Algorithm 8, except that it selects a different corner of the "query" rectangle, o_3, by fixing *selector* to 0 (i.e., selecting the *same* corner as c.mask). Whereas a query checks whether an entire rectangle is contained in the dead space pruned by a single clip point, an insertion checks whether any part of a rectangle is contained in that dead space. If the intersection test returns `FALSE`, then the newly inserted object dominates the clip point (i.e., part of the object is prunned away by the clip point) and thus is invalid. (Inserts propagate up from the leaves, so always $Q \cap R \neq \emptyset$.) If the intersection test returns `TRUE`, the top-k clip points are unchanged and the CBB does not need to be updated.

4.6 Experimental Evaluation

We incorporate our clipping algorithms from Section 4.5 into a variety of R-tree variants of the existing benchmark for multi-dimensional indexes [16]. The benchmark has been used in many

spatial indexing studies (e.g., [17]). We then evaluate query/update performance, overhead, and spatial join performance relative to unclipped (i.e., unmodified) R-trees.

4.6.1 Environment and Experimental Setup

Spatial indexes. We use `C` implementations of four popular R-tree variants [17], including the quadratic R-tree (QR-tree) [66], Hilbert R-tree (HR-tree) [85], R*-tree (R*-tree) [15], and revised R*-tree (RR*-tree) [17]. We modify each implementation to construct clip points (as per Algorithm 7) for each node prior to flushing the node to disk and to utilise the clip points for intersection tests (as per Algorithm 8). We configured each index using the values for min and max node capacities (M and m) as described in [17]. We observe minimal effect from varying k and τ; they are set to values of $k = 2^{d+1}$ and $\tau = 2.5\%$. (Figure 4.10 illustrates the effect of k nonetheless, but we lack space to also vary τ.) To differentiate the two CBBs, we denote the skyline-only approach from Section 4.4.2 as C^{SKY} and the point-splicing approach from Section 4.4.3 as C^{STA}.

Hardware. All experiments use a commodity desktop with a quad-core Intel Core i7-3770 3.4 GHz CPU, 16GB of physical memory, and a 500GB 7200RPM hard disk. The desktop runs Ubuntu 16.04 LTS (kernel version 4.4.0) and the code is compiled with `gcc` (version 5.4.0).

4.6.2 Data Sets and Queries

Data sets. For the main evaluation, we use seven spatial data sets in 2 and 3 dimensions. Four challenging (two real and two synthetic) data sets are taken from the existing benchmark [16]. The real data sets include: `rea02` (a 2d data set of 1 888 012 rectangles and points representing street segments in California) and `rea03` (a 3d data set of 11 958 999 points representing three floating point attributes in a biological data file). The synthetic data sets are `par02` and `par03` containing 1 048 576 2d and 3d boxes, respectively. The objects are generated with a very large variance in size and shape, which makes them challenging to approximate.

In addition to the above data sets, we add three new data sets stemming from our main use case and collaboration with neuroscientists in the Human Brain Project [106]. The data sets contain volumetric boxes representing different spatial objects in a 3d brain model: 2 570 016 segments of axons (`axo03`), 1 288 251 dendrites (`den03`), and 3 858 267 neurites (`neu03`). We show 2d (x-y) projections of the new `axo03` and `den03` data sets in Figure 4.18 (`neu03` is a combination of both). The visualisations give a quick intuition as to how drastically the new data sets differ from any of the 14 data distributions available in [16]. Compared to the rectangular objects in our synthetic data sets and the street segments in `rea02`, the neuroscience objects do not visually suggest being approximated well by (unclipped) MBBs.

In Section 4.7, we extend our evaluation to 28 data sets (also of higher dimensionality), inclusive of those in the main evaluation, in order to reproduce the entire experimental study of [17].

Queries. We query data as in [16]. Given data set D and number of result objects $|R|$ as input, the generator produces queries originating from the dithered centers of the objects in D. $|R|$ object centers are chosen randomly so that the most dense data regions are also most actively queried. For each data set, we produce three query profiles of varying selectivity: *QR0*, *QR1*, and *QR2* retrieve approximately 1, 10 and 100 objects, respectively, per query.

4.6.3 Results and Discussion

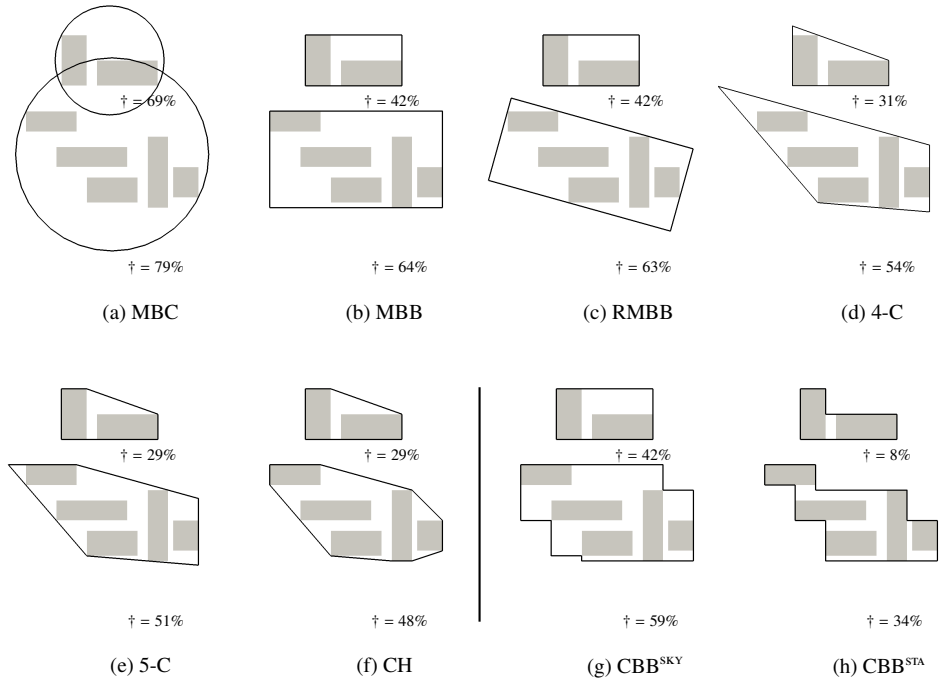


Figure 4.8 – Visualization of different bounding methods over the two leaf nodes from Figure 4.3a and their dead space (†).

Bounding object comparison. We begin by evaluating how well the clip points (CBB^{SKY} and CBB^{STA}) eliminate dead space, relative to six alternative bounding objects (studied earlier in [26, 27]). Figure 4.8 illustrates each shape on our running example. We compute minimum bounding circles (MBC) as per Welzl [163] and rotated minimum bounding boxes (RMBB) by iterating the edges of the convex hull (CH) and computing the minimum bounding box with the same orientation as each edge. The m-corner polygons (4-C and 5-C) are the smallest-area polygons with $\leq m$ corners that fully bound the children, computed similarly to [2]. Finally, the convex hull (CH) is computed using Graham Scan [61]. Observe the extremes that apply in general: CH lower bounds the dead space for all the convex polygons but has the highest representation overhead ($O(n)$ corners). MBC and MBB have the lowest representation overheads ($\leq$ two points) but coarsely approximate their contents.

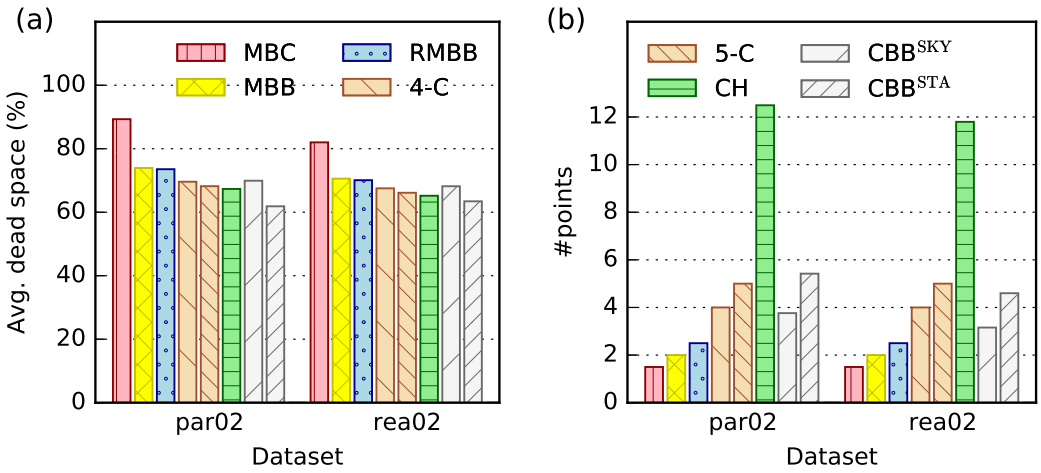


Figure 4.9 – Comparing different bounding methods w.r.t dead space (left) and storage requirements (right).

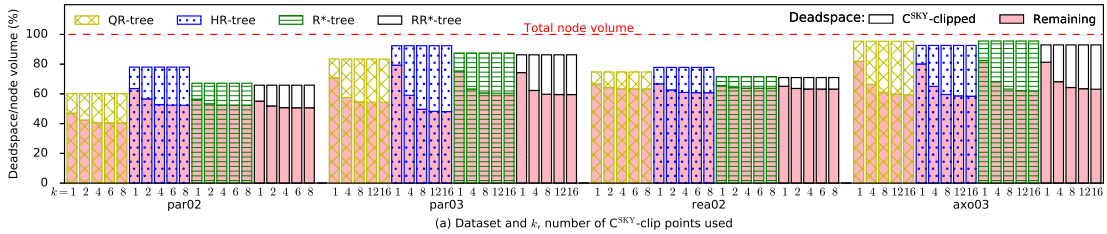
Figure 4.9 contrasts the eight bounding objects in terms of area (left) and representation overhead (right). Following [26, 27], we restrict to $2d$ data sets (just this figure), as we know of no way to calculate minimum bounding m-corner polytopes in higher dimensions. The simplicity with which MBC, MBB, and CBB generalise to higher dimensions is a clear advantage.

For each data set, we built a RR*-tree. The RR*-tree benefits the least from CBBs compared to the other R-tree variants. For each node of the RR*-tree, we replace the MBB with a new minimum bounding shape and measure its area. Figure 4.9(a) reports the percentage of this area that is empty, averaged over all nodes, for each bounding shape. The number of points used is reported in Figure 4.9(b). As expected, the convex polygons prune more dead space as the number of corner points increases. CBB^{SKY} is generally competitive with 4-C using only one or two clip points on average. (The reported cost in Figure 4.9(b) includes the two corner points of the original MBB.) CBB^{STA}, using up to 3.4 clip points on average, outperforms even CH, which uses on average 12.5 and 11.8 points.

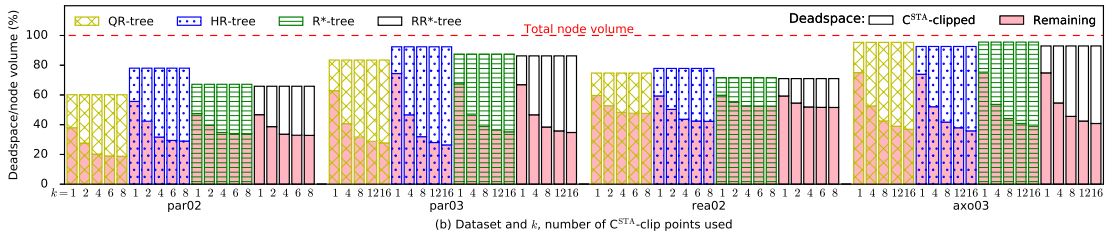
Remaining dead space. Figure 4.10 expands the previous experiment by evaluating the coverage of CBB in all four R-tree variants and more data sets (with some still omitted for space [2]). The total height of each bar, relative to the y-axis indicates the percentage of the MBB that is dead space; the height of the top (clear) part of the bar indicates the fraction of the dead space that is clipped away by the skyline (top) or stairline (bottom) points. If the solid, lower part of the bar is comparably short, then most dead space is eliminated.

Along the x-axis, we vary three parameters: the highest granularity groups correspond to a given data set; the four groups within that are colour- and pattern-coded by R-tree variant; at the

[2]The point-only `rea03` data set essentially occupies zero volume; so, the entire MBB is dead space at the leaf node level and the experiment is not very informative. The results for `den03` and `neu03` are very similar to `axo03`.



(a) Skyline-based clipping (described in Section 4.4.2)



(b) Stairline-based clipping (described in Section 4.4.3)

Figure 4.10 – Average dead space per node and R-tree for skyline- (above) and stairline-based (below) clipping.

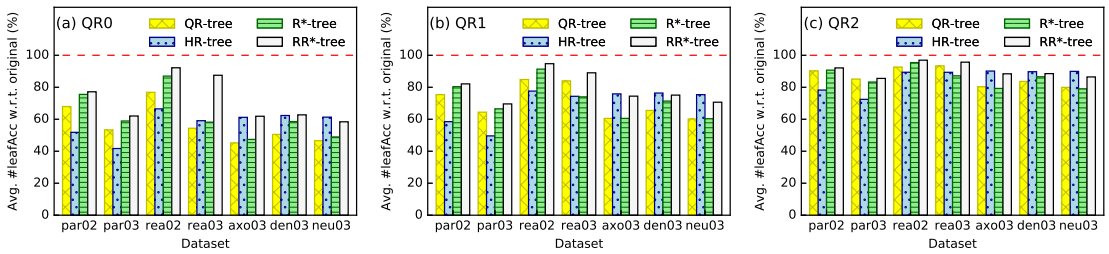
smallest granularity, we vary k, the maximum number of clip points stored per node. We vary k from 1 to 2^{d+1} (i.e., up to twice the number of corners), although reiterate that this is a maximum bound per node: clip points with a score less than $\tau = 2.5$ % are not indexed.

Analysis. We first consider broad trends across data sets. All R-tree variants produce bounding boxes on the neuroscience data set (`axo03`, far right) that are mostly dead space. In fact, for all four data sets, nodes on average contain ≥ 60 % dead space. The increase from the 2*d* synthetic `par02` data set to the similarly generated 3*d* data set illustrates the known fact that bounding boxes become poorer approximations of their contents as the dimensionality increases. Despite different packing algorithms, the QR-tree, R*-tree, and RR*-tree produce similar occupancy rates in their bounding boxes. This is generally less than the HR-tree (the exception being the neuroscience data set where all four have extremely high dead space rates). We observe that most of the dead space is caused by (the much more frequent) leaf-level nodes (not shown independently), which must bound the actual spatial objects using a rough box; at higher levels of the tree, the MBBs more tightly bound other MBBs. Overall, such voluminous dead space across R-tree variants and data sets strongly motivates this research.

Turning to the ratio of clipped dead space to remaining dead space (i.e., fraction of each bar that is filled in), we see more variation across data sets, but consistency across R-tree variants. It is more difficult to clip away dead space from the street segments (`rea02`), which is quite intuitive: we expect street segments to "wrap around" some of the dead space, particularly in cities with grid patterns. Nonetheless, even on this least promising data set, we clip away more than a fifth

	QR0	QR1	QR2	Total
QR-tree	24/44	16/29	7/13	16/29
HR-tree	25/42	18/30	8/14	17/29
R*-tree	21/38	15/28	7/14	14/27
RR*-tree	15/28	11/21	4.5/9.5	10/19
Total	21/38	15/27	6.5/13	14/26

Table 4.1 – Average improvement in % I/O reduction using skyline/stairline clipping for each R-tree.



(a) QR0 **(b) QR1** **(c) QR2**

Figure 4.11 – Average #leaf accesses in clipped R-trees w.r.t. their unclipped counterpart (100%) for stairline-based clipping.

of the dead space. For the 3d data sets, we clip away more than 60%, irrespective of the packing method.

Perhaps most encouraging is the fraction of dead space pruned by the first (ordered) clip point (i.e., at $k = 1$). With just one point, 26%, 23%, 22%, and 22% of dead space is clipped away by QR-tree, HR-tree, R*-tree, and RR*-tree, respectively (on `rea02`). Since we order clip points by the (heuristic) volume of dead space that they clip away, the effect of each subsequently added clip point diminishes and eventually flattens out. Nevertheless, we observe that with $k = 2^d$ the CBBs still eliminate substantial portions of dead space. This suggests that k can certainly be large enough to produce one clip per corner. Overall, clipping with up to two points per corner (i.e., $k = 8$ in 2d and $k = 16$ in 3d), eliminates almost half of all dead space: 58%, 60%, 49%, and 48% is clipped away in QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. Since the number of actually stored clip points is often lower (recall $\tau = 2.5\%$), we set $k = 2^{d+1}$ to this maximum bound in the following experiments.

To compare skyline- and stairline-based clipping, observe that the difference lies in the fraction of the bars that are filled in; the total height (i.e., amount of dead space) is dependent only on the data set and packing algorithm. It is clear, as asserted in Section 4.4.3, that the stairline points clip away much more dead space, eliminating on average almost 50% more for the same k.

Range query performance. While the first experiments show that we clip away a lot of dead space, the real objective is to improve query performance. Removing dead space is only useful if

that is where the query rectangle intersects the MBB. Here we evaluate how well clipping reduces I/Os. Following numerous studies on disk-based indexing (e.g., [17]), we assume that internal (non-leaf) nodes are memory-resident and measure the number of leaf-level nodes accessed as our default I/O metric. (Later we remove this assumption.)

Figure 4.11 reports I/Os for stairline-based clipped R-trees. Query selectivity decreases in the subfigures from left to right. Within each subfigure, four vertical bars corresponding to each R-tree variant are grouped by data set. The bar height reports the percentage of I/Os relative to not clipping.

Analysis. We focus on stairline clipping, but similar (dampened) trends apply to the skyline points. For example, skyline/stairline points clip away 20%/39% of RR*-tree volume, reducing I/Os by 10%/19%, i.e., C^{STA} performs $\approx 2\times$ better.

On the most selective queries (a), we consistently reduce I/Os by $\geq 10\%$, even exceeding 50% in some cases. The gains are particularly pronounced on the neuroscience data sets (`axo03`, `den03`, and `neu03`), where we save 40–50 % of I/Os. While the difference in the amount of dead space clipped away in Figure 4.10 was relatively small between the HR-tree and the other R-tree variants, here we observe a much larger difference. The RR*-tree gains somewhat less query performance from clipping than the other variants (19% versus 27–29% as an average over all query profiles). We attribute that to its already strong query performance (shown in [17]).

The observed performance gains diminish with decreasing query selectivity. This is expected, as the fraction of spatial objects that are around the query boundary decreases with larger query ranges (i.e., lower selectivity). Nevertheless, HR-tree and QR-tree still benefit appreciably (circa 20%) on the `rea02`/`rea03` and neuroscience data sets, respectively. The RR*-tree gains relatively less than the other variants as query selectivity decreases, and on QR2, the difference between variants is not very pronounced.

Table 4.1 averages the performance gains of Figure 4.11 across all data sets. For QR0, average relative I/Os drops to 56%, 58%, 62%, and 72% for QR-tree, HR-tree, R*-tree, and RR*-tree, respectively. In total, clipping MBBs results in an average reduction of 26% in I/Os, considering all data sets, query profiles, and R-tree variants.

Update cost. Next, we quantify how effective are the Section 4.5.4 strategies for avoiding unnecessary re-clipping of CBBs. Recall that we never re-clip on a deletion if the MBB is not changed, so this experiment focuses on insertions.

We first randomly choose 90 % of the input file to batch-construct the clipped R-tree variants. Then, we execute our insertion routine for each of the remaining 10 % of objects. We report on the *y*-axis of Figure 4.12 the *expected number of re-clips per insertion*: i.e., the number of nodes that we re-clip divided by $0.1\times$ the input file size. Along the *x*-axis, we vary data set and R-tree variant. Each stacked bar shows the cause of the re-clip: at the bottom are node splits, which always force an MBB recomputation. In the middle are MBB changes without a node split,

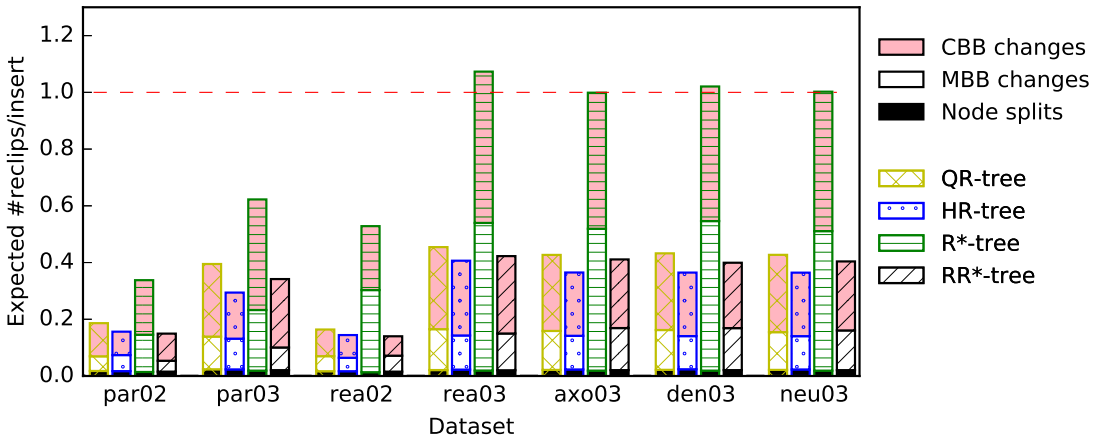


Figure 4.12 – Expected number of re-clipped CBBs per insertion.

which force a CBB change. The top part shows CBB changes with no corresponding change to the MBB, i.e., when Algorithm 8 returns `FALSE`.

Analysis. Without the strategy for avoiding unnecessary re-clips presented in Section 4.5.4, the number of CBB changes per insert would be exactly 1.0 higher than the number of MBB changes. However, we observe far fewer re-clips than this worst case. Averaging across all data sets, ≤ 0.35 of a node needs to be re-clipped per insert, with the exception of the R*-tree (discussed below). Generally, $\approx$½ the re-clips are caused by underlying MBB changes. The challenging neuroscience data sets give the highest re-clip rates, where we still avoid ≈ 60 %.

With respect to d, the $2d$ data sets have lower re-clipping rates than their $3d$ counterparts (observe `par0*` and `rea0*`). The neuroscience data sets, with long, thin objects, have similar re-clipping rates to `rea03`. The R*-tree consistently suffers the most re-clips because of its reinsertion policy: a node split re-inserts every entry of that node; thus, single-object inserts often cause many MBB/CBB changes.

Time complexity for updates. To quantify the in-memory computation overhead during insertion into the clipped R-trees, we configure buffer sizes so that the R-trees are memory-resident and then measure the resultant insertion time (of 10% of objects as in Figure 4.12).

Figure 4.13 shows the results. While for $2d$ data sets, the overhead is negligible, $3d$ updates are at least $\times 2$ more expensive in clipped variants. The trends also comply with the results in Figure 4.12: the extra (though greatly reduced by our proposed algorithm) CBB overhead (red portion of the bars) still dominates the other costs.

We note that this work focuses on a static setting, driven by the neuroscience use case where the entire $3d$ brain model is built on a supercomputer and dumped to file(s) prior to being investigated by analytical queries. To support more dynamic workloads, most likely we would rely on batching

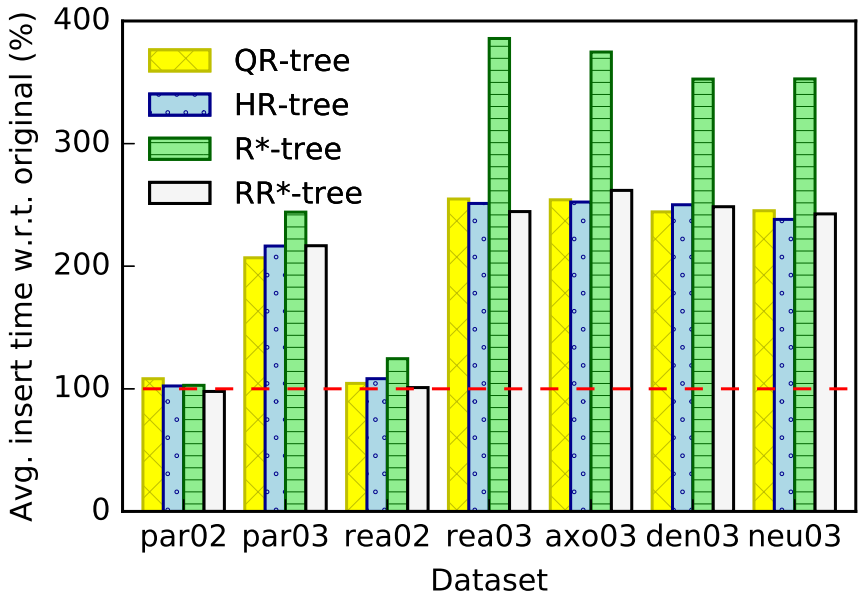


Figure 4.13 – Average update runtime in clipped R-trees w.r.t. their unclipped counterparts (100%) for stairline-based clipping.

or lazy update strategies that aim to make vanilla R-trees more dynamic [21, 93, 170]. All these techniques are applicable to our clipped variants but we have not investigated them in this work.

Extra storage cost. Figure 4.14 reports the increased storage requirements for clipped RR*-trees ($k = 2^{d+1}, \tau = 2.5\%$). Each bar decomposes the percentage of bytes devoted to directory nodes, leaf nodes, and clip points. For each data set (x-axis), we show results for C^{SKY} (left) and C^{STA} (right). As we only retain clip points with scores $\geq \tau$, we report atop each bar the average number of clip points that are stored.

Analysis. Overall, the CBB overhead is quite low and storage is dominated by the far more frequent leaf nodes. The storage dedicated to clip points never exceeds 2% (2d data sets) nor 9% (3d data sets), irrespective of which method (skyline or stairline) is used to generate them. This confirms that clip points and internal nodes can generally be memory-resident, as they contribute just a few percent of the total storage.

No data set averaged all $k = 2^{d+1}$ clip points. As few as 6 (C^{SKY}) and 13 (C^{STA}) clip points are stored per node in the 3d neuroscience data sets; the 2d data sets have ≤ 3 clip points on average. This reflects that some objects are often near to MBB corners; dead space is not uniformly distributed.

C^{SKY} produces fewer clip points than C^{STA} as its clip points prune less area, often $< \tau$. Thus, C^{SKY} has a smaller overhead than C^{STA} (3-fold on `rea03`), but worse I/O performance.

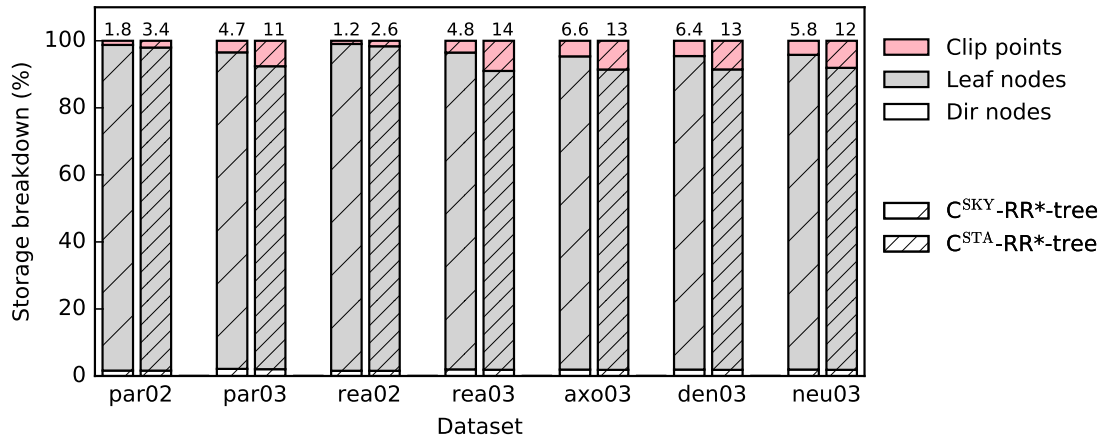
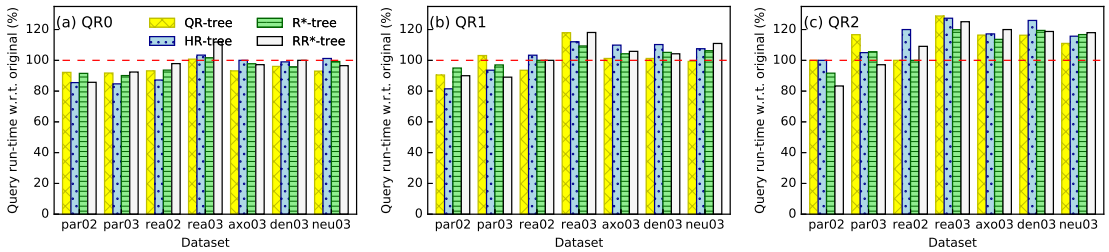


Figure 4.14 – CBB storage overhead.



(a) **(b)** **(c)**

Figure 4.15 – Average query runtime in clipped R-trees w.r.t. their unclipped counterparts (100%) for stairline-based clipping.

CPU overhead in querying CBBs. To quantify the in-memory computation overhead in querying CBBs, we configure buffer sizes for all R-trees so that they are completely memory-resident and then measure the resultant querying time. Results are shown in Figure 4.15. For each stairline-based clipped R-tree, this figure reports the querying time relative to the unclipped variant (100%) on the y-axis. Data sets are varied along the x-axis. The reported CPU overhead is at most 23% for all data sets. In some cases (e.g., in $2d$ data sets under QR0 and QR1 query profiles) querying CBBs is cheaper, even in this in-memory setup, due to better pruning already at the intermediate level nodes.

We note that efficient main memory computation is not the focus of this work, i.e., the current implementation lacks in-memory optimizations. For example, we observe that the average node capacity utilization across the R-tree variants is around 70%. The unused 30% is more than enough to store all node's clip points. This way, most of the time we could avoid chasing a pointer to our auxiliary data structure and thus save at least one cache miss. Additionally, current CPUs support wide SIMD instructions that enable to perform dominance tests between a query corner and multiple clip points at once (data parallelism). Finally, when the MBB is fully covered

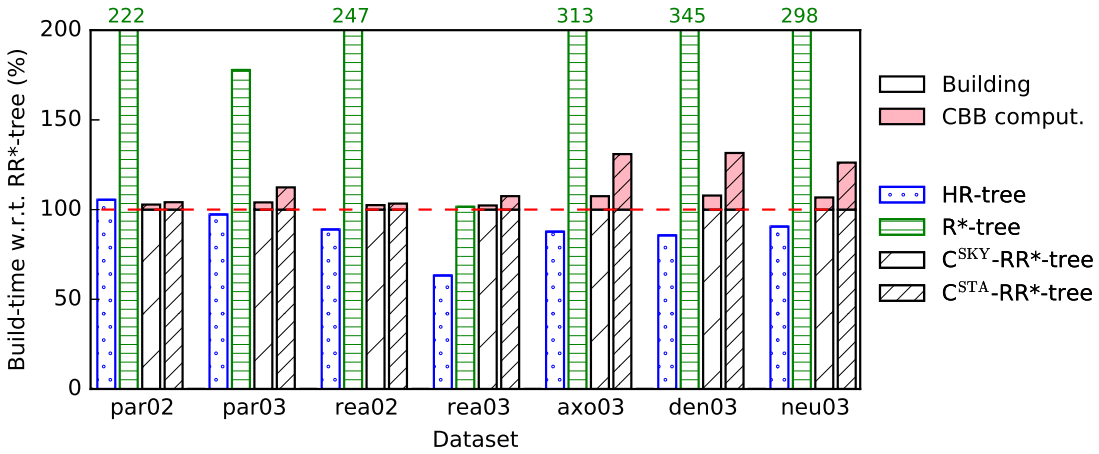


Figure 4.16 – Index building and CBB computation overhead.

by a range query (which is likely for large queries, e.g., the QR2 profile), the CBB intersection tests become unnecessary and can be skipped completely.

Clipping CPU cost. During construction, we recommend to clip an R-tree node in main memory just before flushing it to disk so that the CPU cost is masked by that of the disk write. Nevertheless, for completeness, we quantify the main memory overhead for clipping. To do so, we configure buffer sizes for all R-trees so that they are completely memory-resident and then measure the CPU time.

Figure 4.16 reports the CPU construction time relative to an unclipped RR*-tree (100%). Data sets vary along the x-axis. The HR-tree is generally the fastest to build (due to its bulk-loading) and the R*-tree is generally the slowest to build (due to its forced reinsertion of items during node split). These two unclipped variants provide context for interpreting the clipping overhead. The shaded part (above the dashed line) of the stacked bars for the clipped RR*-trees shows the component of the construction time devoted to clipping. When exceeding 200%, the value is written above the bar.

Analysis. As k grows exponentially with d, so does the overhead of CBB computation. For the spatial data sets (i.e., $d \leq 3$), C^{SKY} adds $< 7\%$ extra computation time. C^{STA} clipping is more expensive, adding up to 4% and 30% of extra computation in $2d$ and $3d$ data sets.

Spatial join performance. We perform a join using `axo03` and `den03`, resulting in 1 985 969 pairs, using stairline points and two join strategies: Index Nested Loop Join (INLJ) and Synchronised Tree Traversal (STT).

Analysis. In the INLJ evaluation, we build an index on the larger data set (`axo03`) and probe it with every object from `den03` (essentially one range query per `den03` object). The results mirror

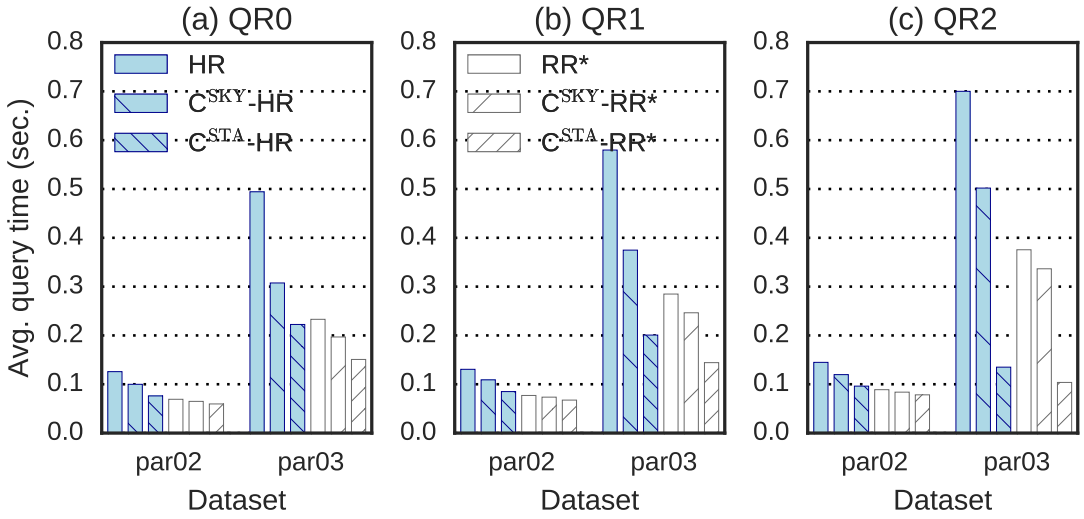


Figure 4.17 – Querying 1 billion object data sets.

those of the range query experiments: clipping reduces I/Os by 40%, 53%, 50%, and 39% in the HR-tree, QR-tree, R*-tree, and RR*-tree, respectively.

For the STT strategy, we measure leaf accesses for both trees and observe a 17%, 20%, 20%, and 16% reduction in I/Os in the HR-tree, QR-tree, R*-tree, and RR*-tree, respectively. Note that the obtained reduction is lower than in the case of INLJ, but STT performs significantly better than INLJ, having a lower total number of accesses. E.g., in the case of the RR*-tree, INLJ with clipping incurs around 4×10^6 leaf accesses whereas STT with clipping only around 10^6. Intuitively, the intersect area of two tree nodes in STT is larger than the intersect area of a tree node with a single data object in INLJ and thus there is a lower chance that it will be fully enclosed within dead space.

Scalability experiment. Our last experiment scales up the synthetic data to 2^{30} objects so that it exceeds our machine's 16 GB of physical memory (yielding 71 GB and 96 GB for the `par02` and `par03` RR*-tree index disk dumps, respectively, with similar sizes for HR-tree). Starting with all indexed data on disk and nothing buffered we measure the query time for 500 random queries on each query profile, allowing the OS to cache paths for previously touched nodes. The average query runtimes are reported in Figure 4.17 for each selectivity.

Analysis. Both skyline and stairline CBBs boost query performance in the HR-tree and R*-tree. Similarly to the I/O performance in Figure 4.11, C^{STA} clipping is 2× more efficient than C^{SKY} on average. Interestingly, a C^{STA}-clipped HR-tree matches (in QR0 and QR1) or even outperforms (in QR2) an unclipped RR*-tree. In all, spatial search with CBBs, even for 1 billion objects, reaches interactive times—200 ms or less.

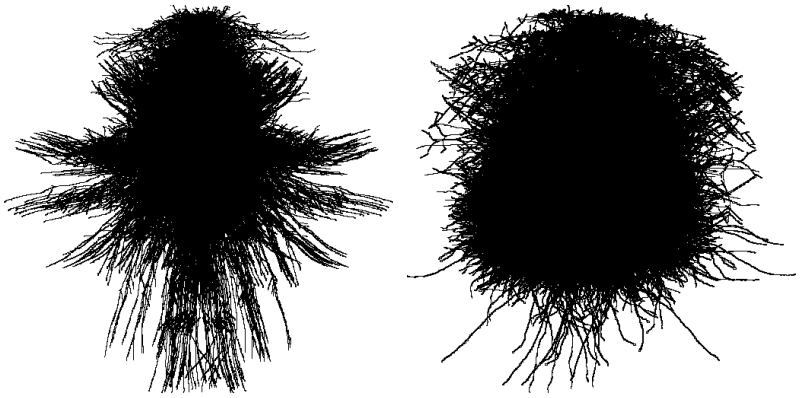


Figure 4.18 – $2d$ **projections of the** $3d$ **`axo03` and `den03` data sets.**

4.7 Reproduction of Prior Experimental Results

This section presents the reproduction of all the experiments given in "Figure 4" of [17] using the data sets and queries shared by the authors. There are 28 data sets in total: 21 generated synthetically and 7 drawn from real-world applications. There are 7 groups of synthetic data sets using different data distributions: absolute (`abs`), power-law (`bit`), diagonal (`dia`), parcel (`par`), p-edges (`ped`), p-haze (`pha`), and uniform (`uni`). Each group contains a $2d$, $3d$ and $9d$ data set that follows the same data distribution. The real (`rea`) data sets correspond to $2d$, $3d$, $5d$, $9d$, $16d$, $22d$ and $26d$ distributions. As before, distribution and dimensionality are encoded into the data set name, e.g., `uni02` and `rea05` stand for $2d$ uniform and $5d$ real data sets, respectively.

For each data set, three query profiles, QR0, QR2, and QR3, are synthetically generated to retrieve ≈ 1, 100, and 1000 objects, respectively, producing a total of 84 query files. Each query file follows the corresponding data set's distribution (except `ped` where queries are generated uniformly to intentionally touch the empty parts of the data space, i.e., to mimic exploratory queries). A detailed specification of all data sets and queries, with visualisations of the data distributions, can be found in [16].

The reproduced results are shown in Figure 4.19. The bottom horizontal axis represents a query index (file), while the top axis additionally labels the data sets queried. For each data set, there are 3 ticks on the bottom axis corresponding to 3 query files (in order QR0, QR2, and QR3). The vertical axis shows the average number of leaf accesses (in percent) in relation to that of the original RR*-tree which is set to 100% for each query file. We exclude query performance for very high-dimensional real data sets (`rea16`, `rea22`, and `rea26`), i.e., queries at indexes 76–84. Instead, we show the performance of the two new real $3d$ data sets (detailed in Section 4.6.2).

In addition to our skyline and stairline clipped variants of the Hilbert R-tree (C^{SKY}- and C^{STA}-HR-tree) and RR*-tree (C^{SKY}- and C^{STA}-RR*-tree), Figure 4.19 depicts query performance of the four

original indexes: the quadratic R-tree [66], the R*-tree [15], the original RR*-tree [17], and the Hilbert R-tree [85]. To make Figure 4.19 less cluttered, we plot the performance of our clipped R-tree variants only if it is at least 5% better in performance compared to its original counterpart. (For example, no red markers for any of the clipped RR*-tree variants from query 1 to 27 implies that clipping improvement is below 5% under these data sets/queries.)

We have obtained the same query performance for all four indexes as reported in the original study [17]. We observe that the RR*-tree is superior for most of the data sets and query profiles. Interestingly, though, this is not the case under our newly added three data sets. The RR*-tree is outperformed by the HR-tree under all new queries (at indexes 76–81, far right) with the biggest gap on query 82 (by 37%).

Our clipped variants do not offer large performance gains for most synthetic data sets except for the more complex distributions in `par02`/`par03`, `ped02`/`ped03`, and `pha09`. Also, as mentioned above, query files for `ped` are generated uniformly (and thus touch a lot of dead space) and both clipped variants are able to benefit from that compared to the original counterparts.

We re-observe two trends from Section 4.6. First, as expected, the C^{STA} variant always outperforms C^{SKY}. Second, smaller queries yield higher performance gains, i.e., QR0 benefits more than QR2 which benefits more than QR3. This is expected as larger query ranges have a smaller fraction of objects located at the query boundaries.

4.8 Chapter Summary

Minimum bounding boxes (MBBs) are ubiquitously used in spatial indexing to represent a set of spatial objects. However, they often enclose significant "dead space" that contains no actual objects. This chapter proposed *clipping away* empty corners of MBBs with a lightweight overhead. Each auxiliary "clip point" defines a large, empty rectangular area that can be discarded with a single point comparison. The resultant bounding shapes are simple but non-convex, thereby pruning more area than previously proposed alternatives to MBBs.

We plugged our skyline- and stairline-based clipping strategies into four R-tree variants of a well-known experimental benchmark. Compared to unclipped R-trees, the (2×) more aggressive stairline points removed ≥ 27% of the dead space. Irrespective of R-tree variant, this translated into ≈ 26% I/O reduction on average across all workloads. With a storage overhead of a few percent, clipped bounding boxes are highly effective for accelerating spatial data processing.

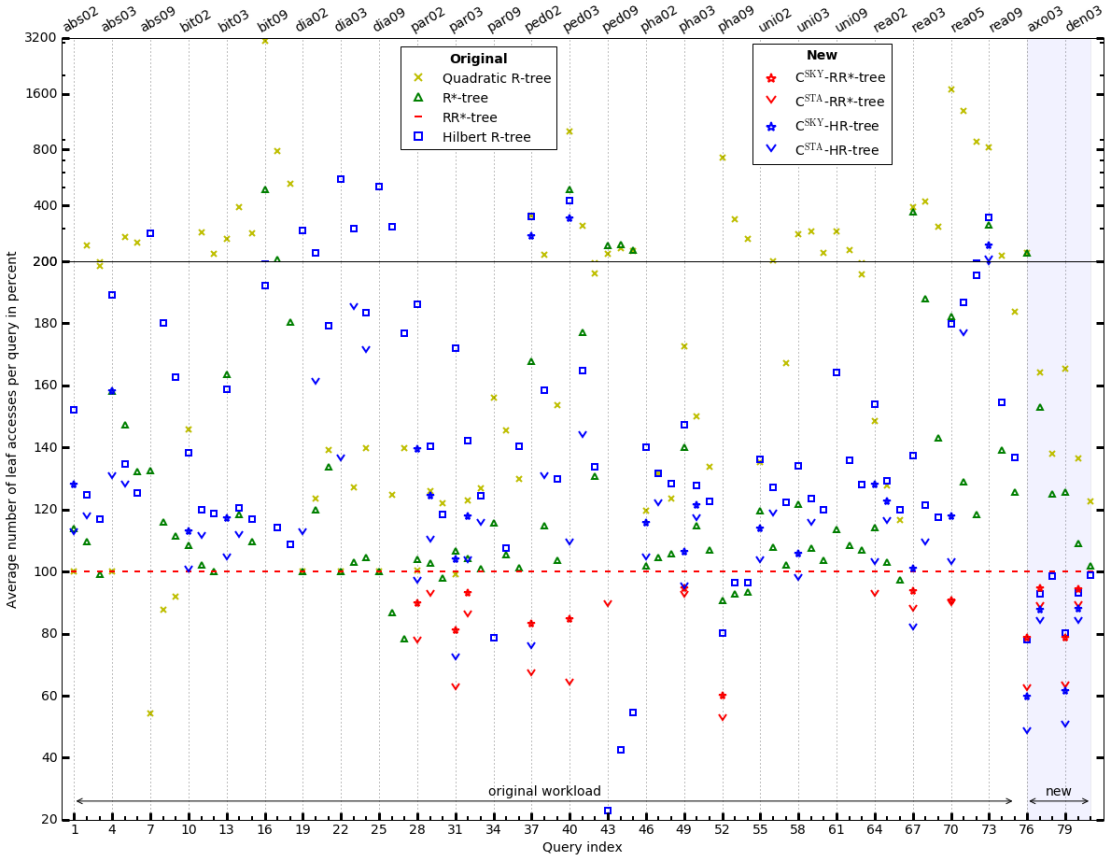


Figure 4.19 – Reproduced "Figure 4" from [17] augmented with our workloads.

5 Workload-Aware Indexing for Ad-hoc Spatial Data Exploration

Efficiently querying multiple spatial data sets is a growing challenge for scientists. Astronomers query data sets that contain different types of stars (e.g., dwarfs, giants, stragglers) while neuroscientists query different data sets that model different aspects of the brain in the same space (e.g., neurons, synapses, blood vessels). The results of each query determine the combination of data sets to be queried next. Not knowing a priori the queried data sets makes it hard to choose an efficient indexing strategy.

In the first part of this chapter, we show that indexing and querying the data sets separately incurs considerable overhead but so does using one index for all data sets. We therefore develop STITCH, a novel index structure for the scalable execution of spatial range queries on multiple data sets. Instead of indexing all data sets separately or indexing all of them together, the key insight we use in STITCH is to partition all data sets individually and to connect them to the same reference space. By doing so, STITCH only needs to query the reference space and follow the links to the data set partitions to retrieve the relevant data. With experiments we show that STITCH scales with the number of data sets and outperforms the state-of-the-art by a factor of up to 12.3.

In the second part of this chapter, we argue that in some cases indexing all the data upfront is unlikely to pay off because scientists do not need to analyze it all. Nevertheless, tools and methods to analyze only data subsets are rather rare. We therefore present Space Odyssey[1], a novel approach enabling scientists to efficiently explore multiple spatial data sets of massive size. Without any prior information, Space Odyssey incrementally indexes the data sets and optimizes access to the ones frequently queried together. As our experiments show, through incrementally indexing and adapting the data layout on disk, Space Odyssey accelerates exploratory analysis of spatial data by substantially reducing query-to-insight time compared to the state-of-the-art.

[1]The material of this part has been the basis for the ExploreDB 2016 paper *Space odyssey: efficient exploration of scientific data* [125].

5.1 Part I: An Index Structure for Multiple Spatial Data Sets

5.1.1 Introduction

In several real-life applications data is naturally divided into distinct categories and users are often interested in a subset of them. Additionally, users issue queries that explore different combinations of categories as they rarely know a priori which categories need to be combined to answer a particular question or test a specific hypothesis. To build an anatomically accurate spatial atlas of the human brain, for example, the neuroscientists in the Human Brain Project (HBP) [106] study the structure and shape of neurons using bright-field microscopy, and whole brain images using magnetic resonance imaging (MRI). This process results in different data sets that contain data originating from different observational sources, potentially representing different types of neurons or other brain structures (e.g., synapses and blood vessels). Inspecting the same brain regions in these data sets allows to verify that a given region contains the correct ratio and distribution of different brain structures, and is key to building a brain atlas. There is consequently the need to efficiently retrieve the same spatial region from a number of different data sets, stored on disk due to their size. This problem is challenging because the queried data sets are chosen ad-hoc depending on the results of previous queries and thus cannot be predicted. As more observational sources are added, the problem becomes increasingly challenging.

Formally, let D be a set of N spatial objects. Each object has a spatial extent[2] and a tag that denotes the object's *category*. Categories have application-specific semantics; for example the different categories can correspond to neurons of different types, or to the same neurons obtained from different samples.

Definition 1 (Bundled Spatial Range Query). Let c be the number of distinct categories; each tag thus is represented as an integer in $\{1, 2, ..., c\}$. Denote by D_i $(1 \leq i \leq c)$ the set of objects in D having category tag i. Every object has exactly one tag, and thus $D_1, D_2, ..., D_c$ are mutually disjoint and $\bigcup_{i \in c} D_i = D$. Given an axis-aligned range query r defined as a three dimensional interval $r = [l_1, u_1] \times [l_2, u_2] \times [l_3, u_3]$ and a non-empty set $Q \subset \{1, 2, ..., c\}$, a bundled spatial range query returns, for each $i \in Q$, all objects $d \in D_i$ intersecting with r. We call the query parameter Q a category selection. Note that Q can be any non-empty subset of $\{1, 2, ..., c\}$, i.e., a combination of categories. The total number of possible Q is $2^c - 1$.

Existing spatial indexing approaches can be applied, but not knowing a priori which combination of categories will be queried together renders them inefficient. Using a single index for all categories is only efficient when a query is executed on all categories. Otherwise the I/O overhead can be considerable as data not belonging to the categories of interest needs to be retrieved from disk and filtered out. On the other extreme, using multiple indexes, one for each category, becomes inefficient when the number of categories is large, as the same spatial region has to be repeatedly located within different index structures (that typically suffer from over-coverage, i.e., dead space, and overlap of minimum bounding boxes [66]). Overall, the dominant cost of using

[2]The description in this paper focuses on 3D objects but the proposed techniques also work on 2D data.

(i) a single index is retrieval and filtering of unnecessary data and (ii) multiple indexes is repeated traversal of index structures.

To achieve the best of both worlds, we introduce a new indexing approach that exhibits superior performance by eliminating unnecessary I/O operations. Our approach consists of two phases. The first phase is category-oblivious; it simply uniformly divides the reference space that encompasses all the categories. The second phase is category-aware and segments each individual category into partitions in a data-driven manner. To enable pinpoint access to regions in the queried categories, the category partitions are linked to the uniform partitions of the reference space. With this strategy, our approach retrieves from each category precisely the data needed without incurring the undue overhead of querying separate indexes.

Note that we target scientific use cases where all the raw data (or at least most of it) is available before querying. We thus focus on developing a bulkloading approach.

Contributions. To the best of our knowledge, we are the first to study in-depth the problem of bundled spatial range queries and to identify the lack of a solution that provides efficient access to individual categories (or data sets) that are all enclosed within the same spatial volume. Given the importance of this problem in real-world applications, we advocate that specialized efforts are required to improve the performance of existing spatial indexing approaches. The main contribution of this work is the development of a spatial indexing approach that scales with an increasing number of categories, yet without penalizing performance when the number of categories is small.

The design of our approach is based on the key observations that most overhead in baseline approaches originates from (a) traversing (hierarchical) index data structures repeatedly and unnecessarily and (b) from late pruning results from unnecessary categories.

We therefore develop an index which is based on a simple, flat, grid-based reference space. With this, query execution can quickly assess which areas are likely to be in the query result without traversing one or multiple hierarchical index structures repeatedly. Furthermore, to enable early pruning irrelevant categories in the query execution, we store information on where and how to retrieve category specific data within the grid, in a data structure that enables efficient filtering based on categories. As a result, our approach scales with an increasing number of categories, avoiding the retrieval of an excessive amount of index related data structures as well as irrelevant results. Our approach also allows to incrementally add new categories with only limited updates to the reference space (and no updates to other categories) because each category is treated, partitioned and stored independently.

To showcase these techniques, we develop STITCH, a bundled spatial index that achieves efficient query execution while scaling with an increasing number of categories. Albeit we base STITCH on simple ideas, these ideas prove to be effective. As our experiments on real-world data show, STITCH achieves a speedup of $\sim 4.5\times$ compared to indexing each category separately with a state-of-the-art index, and $1.3\times$ - $12.3\times$ compared to indexing all categories with a single index.

5.1.2 Motivation

Spatial data is at the core of many scientific processes as scientists study entities using their morphological or topological properties. Observational data is acquired using a variety of different instruments and techniques and originates from a variety of input samples, resulting in multiple data sets describing the same spatial volume. Given these data sets, scientists need to efficiently perform ad-hoc queries collecting data from only a subset of them. Querying a certain combination of data sets that are arbitrarily chosen from a pool of tens or hundreds of data sets is the general problem that drives the design of STITCH.

Use Cases. The main motivation behind our work stems from our collaboration with the Human Brain Project (HBP) [106]. Neuroscientists in the Human Brain Project aim to build an atlas of the human brain which will serve as a unifying "spatial scaffold" for studying different aspects of the brain. The input to create this atlas is observational data collected using a variety of light microscopy modalities (such as laser-scanning, wide-field epifluorescence, and bright-field microscopy) and magnetic resonance imaging (MRI). Scientists then need to ensure that their spatial model is biorealistic. To do so, they compute statistical properties for different regions in the model and compare them with the observational data. Overall, in both the building and the validation phase, scientists need to retrieve complementary information about a given spatial region from a subset of the data sources that are available to them in an exploratory fashion. The bulk of the data used in this analysis is static, as scientists incorporate new information by adding new data sources rather than updating existing ones. With the increase in the number and size of the analyzed data sets, the exploration process is significantly hampered.

In other disciplines (e.g., cosmology [92] and seismology [5]) scientists simulate phenomena on a large scale. The outcome of the simulations are several data sets, each containing a different representation of the simulation result and, in the analysis of the result, parts of different data sets need to be combined. In cosmology, for example, N-body simulations of different particle types (dark matter, gas, stars, etc.) are used to study the evolution of the universe. Each resulting data set stores the locations of one particle type and for the final analysis, astronomers need to query different data sets together without knowing a priori the exact combination.

Data Management Challenge. There are three straightforward strategies for evaluating bundled spatial range queries using existing spatial indexes. The first strategy, `1-for-each`, builds a dedicated spatial index (e.g., R-Tree [66]) for each category. A bundled spatial range query is evaluated by searching the $|Q|$ spatial indexes on the categories in Q. Adding a new category incurs negligible overhead as it simply entails building an index for the new category. The second strategy, `all-in-1`, builds a single index structure containing all categories. Given a bundled spatial range query, it traverses the index to get a set of spatial objects potentially qualifying the query predicates, it filters out irrelevant items that do not belong to any of the queried categories in Q, and finally evaluates the spatial predicate on the remaining items. New categories are added by updating the index, which can incur substantial overhead. The third strategy, `queried-in-1`, takes the `all-in-1` strategy to the extreme and builds indexes for

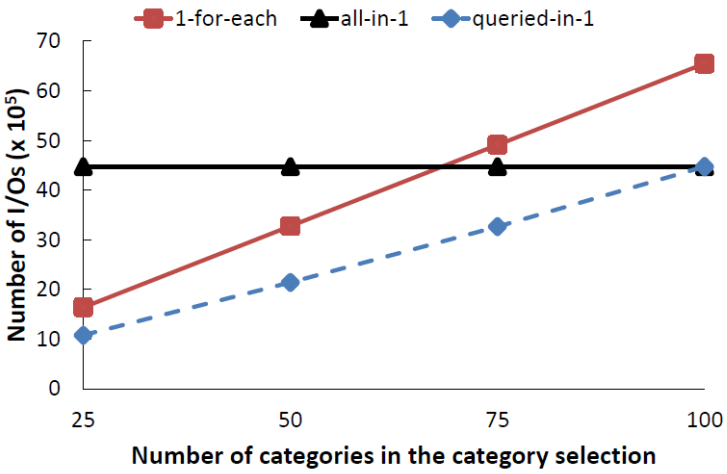


Figure 5.1 – Scaling with an increasing number of categories in the category selection. `1-for-each` does not scale well as the number of categories increases, `all-in-1` introduces an overhead when only a small subset is queried, while `queried-in-1` provides the best performance, but is a practically infeasible solution.

all possible category combinations. That way, we can answer a bundled spatial range query by searching the specific index that contains the exact combination of queried categories (and nothing but those).

We implement all strategies using the R-Tree [66] spatial index, which is arguably the most widely used index structure for spatial data. In a motivation experiment we index 100 neuroscience data sets (categories) representing the same brain volume, $\sim 1GB$ each, and measure the total number of I/Os (i.e., disk pages read) for 200 range queries corresponding to different brain regions of size $10^{-3}\%$ of the total volume. To evaluate the performance of the `queried-in-1` strategy, we create indexes that contain exactly the combination of the 25, 50, 75, and 100 categories that are queried together. The precise experimental setup is described in Section 5.1.7.

The results of the motivation experiment in Figure 5.1 demonstrate the trade-offs of each strategy as the number of categories in the category selection increases. The `all-in-1` strategy achieves the same consistent performance regardless of the number of queried categories, while the performance of `1-for-each` is linearly decreasing with the growing number of categories. This is expected, as the former always has to inspect the same (big) index, while the latter has to probe an increasing number of (smaller) indexes. Interestingly, none of these two strategies achieves the best performance in all cases and the crossing point clearly indicates what strategy is preferred for what scenario. As expected, `queried-in-1` provides the best performance, minimizing the query cost irrespective of how many categories are queried. Note, however, that this strategy is unrealistic because the cost of constructing and storing indexes for all possible category combinations ($2^{100} - 1$ in our example) is prohibitively high.

The goal of this work is to develop an indexing approach that has the same querying behavior as if there was a dedicated index for the queried combination, without building indexes for all possible combinations a priori. As we will show next, this is achieved by physically bundling the indexes for different categories.

5.1.3 Related Work

Data-oriented Partitioning. Arguably the seminal spatial index structure is the R-tree [66]. The R-tree is a disk-based index consisting of a hierarchy of minimum bounding boxes (MBBs) which recursively enclose data objects. By doing so, the R-Tree is resilient to data skew, but faces the problems of over-coverage and overlap of MBBs, which results in multiple (partial) paths being explored during querying. Many extensions to the basic approach have been proposed to address these issues and optimize the node MBBs during dynamic index maintenance. To increase robustness against different data distributions, the R*-tree [15] employs multiple optimization criteria to choose the node into which a new object should be inserted. In addition, it removes and reinserts the spatial objects of an overflowing node in an attempt to minimize the dead space and the margin in each node. The improved query performance comes at the expense of higher update costs. The RR*-tree [17] introduces more adaptive optimization strategies to further reduce I/O costs and enhance search performance. The R+-Tree [131] creates non-overlapping nodes by inserting objects into multiple leaves, which makes the index larger. The cR-tree [23] considers the R-tree node splitting procedure as a typical clustering problem: upon a node overflow, the well-known k-means clustering algorithm is applied to find a good split. To find real clusters, and not just two groupings of the node data, the two-way node splitting property is relaxed. Instead of modifying the index structure or the splitting procedure, the approach presented in Chapter 4 proposes to solve the problem of over-coverage by improving MBBs. It converts node MBBs to CBBs (Clipped Bounding Boxes) by clipping away dead space that is concentrated around the MBB corners.

Since our data sets are massive and known a priori, we focus on bulkloaded R-Trees. Bulkloading approaches group spatially close objects and store them on the same disk page to improve locality and reduce overlap between nodes. Then, an R-Tree is built on top of those disk pages, typically bottom-up. The Hilbert R-Tree [85] uses the Hilbert space-filling curve to order the objects according to their spatial proximity. Sort-Tile-Recursive (STR) [95] recursively tiles the space, sorts the objects in a tile along each dimension and thereby also guarantees spatial proximity as well as small MBBs, outperforming the Hilbert R-Tree [85]. In contrast, the Top-down Greedy Split (TGS) [56] works top down: it splits the data set into partitions so that on each level the area of each partition is minimized. This process continues recursively until each partition fits on a disk page. While bulkloading with TGS takes much longer than with other approaches, the resulting R-Tree outperforms the Hilbert R-Tree and STR on extreme data sets (with respect to skew and aspect ratio). The Priority R-Tree (PR-Tree) [12] groups all objects with extreme coordinates in the same dimension in the same node, thereby reducing the area and overlap of the remaining nodes. This improves performance on extreme data sets, making the PR-Tree outperform TGS.

As recently shown [144], R-Tree-based approaches still suffer considerably from unnecessary I/Os caused by overlap. FLAT [144] consequently adds connectivity (neighborhood) information so that the R-Tree is only used to locate any single data object inside a query volume and the remaining objects are found by crawling through neighbors.

Space-oriented Partitioning. Instead of grouping objects hierarchically based on their proximity and allowing groups to overlap, another family of spatial indexing methods splits the space using hyperplanes into a set of disjoint partitions that are stored flatly [6] or in a hierarchical structure [18, 80, 155]. The simplest space-oriented indexing technique is the uniform grid [6], where a predefined area is divided into rectangular cells. Each cell stores together all the objects that overlap with it. In contrast, the KD-Tree [18] divides the space hierarchically in a data-driven manner. At each level, it splits the objects along one dimension in two partitions such that each partition contains approximately the same number of objects.

Finally, we note that the term *category* in this work simply refers to a *group* of objects (e.g., a data set) rather than a textual attribute. Thus, research on *keyword* search (i.e., [96]) is not related to this work.

5.1.4 STITCH Overview

To overcome the aforementioned challenges, the proposed method STITCH avoids the repeated traversal of multiple index structures on disk and the retrieval of unnecessary data. This is achieved by combining data-oriented partitioning with space-oriented indexing.

First, similar to the `1-for-each` and unlike the `all-in-1` strategy, spatial objects belonging to different categories are stored in separate data files to enable retrieval of data from precisely the categories needed. To retrieve data from each file efficiently, spatially close objects are stored on the same disk page. The assignment of spatial objects to disk pages is achieved by applying a data-oriented partitioning method which adapts to the distribution of objects in each individual category.

Second, unlike the `1-for-each` but similar to the `all-in-1` strategy, instead of using one index per category to index the minimum bounding boxes (MBBs) of the pages, STITCH builds a single index on the reference space (the common universe that encloses all the underlying objects from all categories). To make access to this *reference index* efficient, STITCH avoids a hierarchical structure and organizes it in a uniform grid. The grid cells store links to the categories, i.e., each cell stores links to the category pages it overlaps with. To have more pruning power in the reference index, along each page link, the cell also stores the page MBB. Within each cell, the links (and corresponding page MBBs) are ordered, such that links to a specific category can efficiently be accessed. Specifically, all links for category i precede any link for category j for $i < j$. Figure 5.2 shows the overview structure of STITCH.

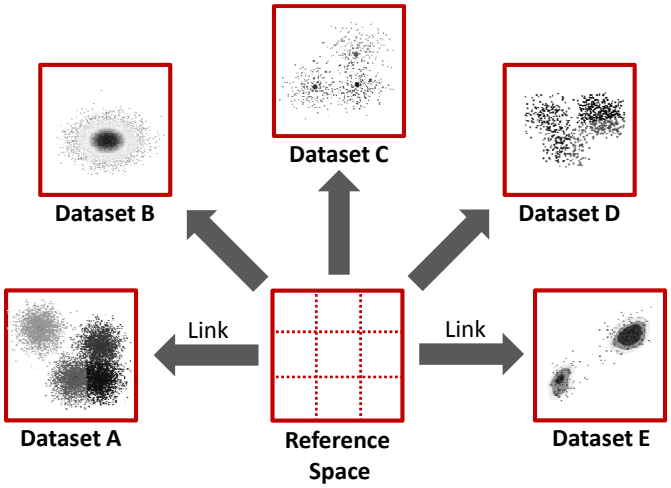


Figure 5.2 – STITCH links multiple data sets (categories) to the same index/reference space (bottom center) and directs queries to the destination data sets via corresponding links.

With the reference index and the category pages, the result of range queries is computed in two phases. STITCH first probes the reference index to find all grid cells overlapping with the query range and retrieves all the page MBBs from those cells for the queried categories. This phase is based on the key insight that unlike data-oriented hierarchical indexes, our reference index does not suffer from overlap. Furthermore, STITCH avoids traversing multiple index structures because all the categories are mapped to the same reference index. In the second phase, STITCH discards the page MBBs that do not fall in the queried area and only visits the qualifying disk pages by following the corresponding links. That way, STITCH retrieves spatial objects from exactly those categories needed and for the queried area, thereby avoiding the retrieval of unnecessary data. Comparing to existing grid-based indexing approaches, STITCH suffers less from the problem of data replication and can thus accommodate categories having objects of varying sizes without the need for expensive fine-tuning of the grid configuration. This is because the uniform grid in STITCH does not index the spatial objects themselves but the page MBBs - the actual spatial objects are organized using a data-oriented space partitioning strategy which is resilient to data skew. As we discuss in more details in the following, we adapt the data-oriented partitioning strategy to reduce the amount of replicated data even further.

5.1.5 STITCH Indexing

We segment the entire space of each category into partitions, each partition corresponding to one disk page. We then create a link between a partition P and a grid cell C of the reference index if and only if at least one element contained in P overlaps C. The created link is essentially the pointer to the partition P (i.e., the location on disk where P is stored) and is stored in grid cell C.

In theory, any partitioning method can be used to partition the categories. Nevertheless, to avoid the problem of replication associated with space-oriented partitioning, STITCH follows a data-oriented partitioning strategy. In particular, we base our partitioning strategy on an existing algorithm, Sort-Tile-Recursive (STR [95]). STR first sorts the spatial objects in the x-dimension and partitions them along this dimension into fixed sized partitions. Each such partition is subsequently sorted and partitioned in the other dimensions (y, and then z). The partition sizes in each dimension are chosen so that the final partitions contain at most as many spatial objects as can be stored on a single disk page.

To minimize partial overlap with the grid cells of the reference index, STITCH extends STR to align the data partitions with the (conceptual) cell boundaries. This is achieved by adding spatial objects in the sorted order to partitions in each dimension until: (i) a grid boundary is crossed, or (ii) the current partition is full. The first condition makes the partitioning grid-aware and minimizes the number of cells that a final category partition overlaps with (thus minimizing the number of replicated links in the reference index). While doing so may underfill disk pages, i.e., less than the maximum elements are stored in a partition, our strategy proves effective in reducing replicated links and providing fine-grained data access. The strategy ultimately trades disk space for performance.

We call our grid-aligned data-oriented partitioning strategy Sliced Data-Oriented Partitioning (`Sliced-DOP`). `Sliced-DOP` has the same complexity as STR, as it does not introduce any additional passes over the data. The pseudocode of `Sliced-DOP` is given in Algorithm 9. Note that the only information needed to detect a grid boundary crossing is the grid resolution, defined by the input parameter g. Also note that the linking is performed alongside the partitioning and it is based on object overlap and not on page MBB overlap: if the intersection between a page and a grid cell does not contain any objects (dead space), the cell is not linked to the page.

Algorithm 9: `Sliced-DOP`

Input: D: array of spatial objects
g: # of uniform grid cells in the reference space
ps: partition size (e.g., # objects per disk page)
Output: P: array of all partitions (stored on disk)
L: reference index (stored on disk)

```
// Init. running partitions for each dimension:
1  P_x ← ∅; P_y ← ∅; P_z ← ∅;
2  C ← ∅ // set of grid cells overlapping a running partition
   // partition sizes:
3  s = ∛(|D|/ps); s_x = |D|/s; s_y = s_x/s; s_z = s_y/s;
4  sortByX(D);                                   // sorts by x-coord.of object centers
5  tile_x ←cellNrAtX(g, D[0]);                   // current tile at x-dim.
6  foreach i ∈ D do
7      nextTile_x ← cellNrAtX(g, i);
8      if tile_x == nextTile_x then
9          P_x ← P_x ∪ {i};
10     if |P_x| == s_x or tile_x ≠ nextTile_x then
11         sortByY(P_x);
12         tile_y ← cellNrAtY(g, P_x[0]);
13         foreach j ∈ P_x do
14             nextTile_y ← cellNrAtY(g, j);
15             if tile_y == nextTile_y then
16                 P_y ← P_y ∪ {j};
17             if |P_y| == s_y or tile_y ≠ nextTile_y then
18                 sortByZ(P_y);
19                 tile_z ← cellNrAtZ(g, P_y[0]);
20                 foreach k ∈ P_y do
21                     nextTile_z ← cellNrAtZ(g, k);
22                     if tile_z == nextTile_z then
23                         P_z ← P_z ∪ {k};
                           // Keep track of overlapping cells:
24                         C ← C∪ cellNr(g, k);
25                     if |P_z| == s_z or tile_z ≠ nextTile_z then
26                         P ← P ∪ {P_z};          // ready partition
27                         foreach c ∈ C do
28                             store in cell c of L the pointer to P_z and the MBB of P_z ; // linking
29                         tile_z ← nextTile_z; P_z ← ∅; C ← ∅;
30             tile_y ← nextTile_y; P_y ← ∅;
31     tile_x ← nextTile_x; P_x ← ∅;
32 return P, L
```

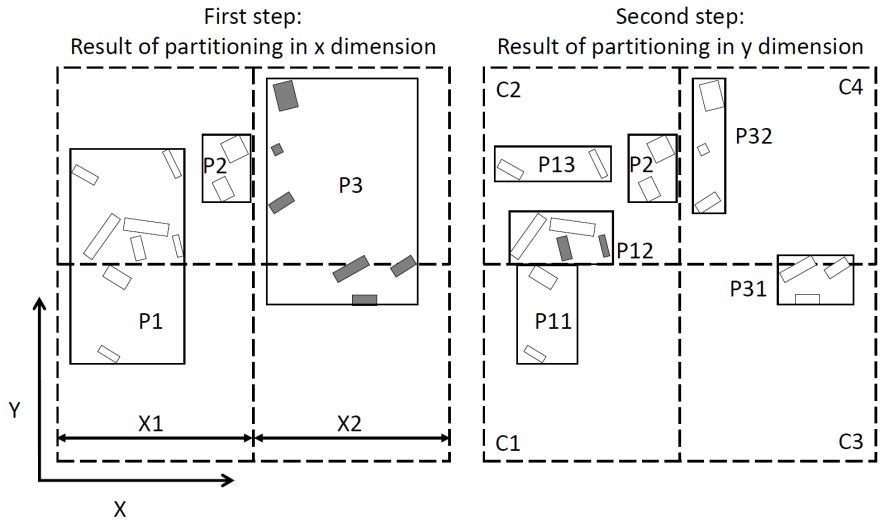


Figure 5.3 – The partitioning procedure packs spatially close elements on the same disk page (rectangle) and aligns the page boundaries as much as possible with the grid boundaries (dashed lines) of the reference index.

Figure 5.3 illustrates the intuition behind our partitioning technique for an example in 2D. This example assumes that each partition contains at most 4 spatial elements — the same number a disk page can store. Since there are 16 objects in total, they are first divided in x-partitions of maximum 8 objects each along the x dimension, and then they are further divided in the y dimension so that the final partitions contain at most 4 objects. The conceptual grid boundaries are shown with dashed lines. When dividing in the x dimension, partition $P1$ contains the maximum number of objects, as it is fully enclosed in a single grid x-tile ($X1$). On the other hand, even though $P2$ is not full, we do not add the next elements in the sorted order (shown with solid color) in it, so that it remains fully enclosed in the first grid x-tile ($X1$). The remaining six elements are then inserted in $P3$. Similarly, when dividing in the y dimension, although $P11$ is not full, we do not extend it with the two elements shown with solid grey color so that it remains fully enclosed in $C1$, and a new page, $P12$, starts in $C2$. The same logic applies in the division of $P3$.

The links introduced in the reference index are the following: $C1$ is linked to page $P11$, $C2$ is linked to pages $P12$, $P13$ and $P2$, $C3$ is linked to $P31$, and lastly $C4$ is linked to $P31$ and $P32$. This example also highlights that in some cases, avoiding link replication is not possible: $P31$ contains two objects that overlap with both $C3$ and $C4$. As a result, both cells need to contain a link pointing to $P31$. The impact of replication, however, is significantly smaller compared to existing grid-based indexing approaches which replicate individual objects. This is because replication happens at the disk page level and typically objects overlapping the same neighboring cells are stored in the same page.

Note that `Sliced-DOP` is *not equivalent* to first applying uniform grid partitioning and then performing STR within each grid cell. The difference is that `Sliced-DOP` sorts the objects

Algorithm 10: STITCH Indexing Algorithm

Input: *datasets*: array of multiple spatial data sets, each corresponding to a distinct category
g: # of uniform grid cells in the reference space
ps: partition size (e.g., # objects per disk page)
ht: in-memory hash table storing the disk offsets of non-empty grid cells

```
foreach D ∈ datasets do
    P ← Sliced-DOP(D, g, ps);
    foreach partition p ∈ P do
        calculate (based on resolution g) the grid cells c that overlap with p;
        store the pointer to p and the MBB of p in each of the grid cells c;
    foreach grid cell c do
        store in the header page of c the number of pointers in c;
        store in the ht the disk offset of the header page of c;
```

globally in each dimension, not *locally* within each grid cell. As a result, `Sliced-DOP` preserves the relative one-dimensional distances between objects across grid cells.

The pseudocode for indexing in STITCH, taking into account multiple categories, is given in Algorithm 10.

Data Structures

This section discusses how we store the partitioning and linking information to support efficient query execution with STITCH. In particular, we discuss STITCH's core data structures, the *metadata* records containing information about each object page, the *reference index* used to retrieve the metadata records in the queried range, and finally *object pages* storing the actual spatial elements.

1) *Metadata:* A metadata record refers to a particular object page and contains a link (pointer) to the object page and the page MBB. By recording the MBB with each link, the data partitions that do not overlap query volumes can be filtered immediately without the need of reading the disk page(s) storing the actual objects.

2) *Reference Index:* To start query execution, the *reference index* must return all the metadata records that fall inside the query range. In STITCH we use a disk-based uniform grid to organize all the metadata records for all the categories in the reference index. A metadata record is stored in a grid cell if the corresponding object page contains at least one object that overlaps with the grid cell. Real simulation data sets have a skewed data distribution and as a result the number of metadata records stored with each grid cell can vary significantly, while a majority of the grid cells are empty. STITCH therefore also maintains an in-memory hash table to store the disk offsets of all the non-empty grid cells. Additionally, the header disk page of each grid cell records the number of links that fall in the grid cell for each category. The links are ordered per category,

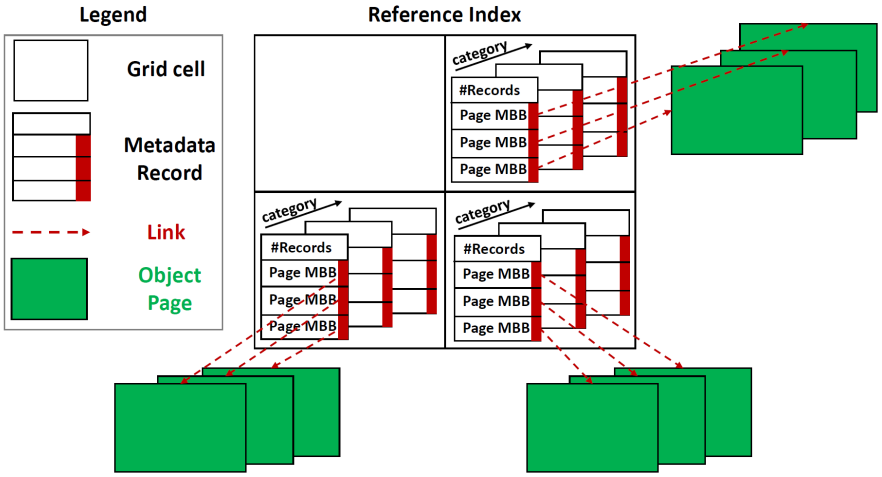


Figure 5.4 – STITCH's data structures and their interaction: The disk-based reference index stores the metadata records in its grid cells which point to the object pages. STITCH also maintains an in-memory hash table indicating the non-empty grid cells which is not shown in the Figure.

so that all links for category i precede any link for category j for $i < j$. The metadata records of each grid cell are flushed to disk sequentially. Spatially close records are very likely to be stored on the same grid cell and thereby on the same disk page, resulting in good disk locality for fast retrieval of the metadata.

3) *Object Pages:* On each disk page, `Sliced-DOP` packs the maximum possible number of elements while at the same time ensuring that in most cases an object page is linked with only one grid cell of the reference index. The exact number of elements that a page can hold depends on their size, e.g. for an axis aligned box with an id the size is 6 floats plus 1 integer. Spatial locality is preserved by storing spatially close objects on the same page.

All data structures and their relations are illustrated in Figure 5.4: several metadata records are stored on each grid cell of the reference index and each metadata record contains a link to an object page.

5.1.6 STITCH Query Execution

STITCH answers a query on a subset Q of all categories in two phases: it first retrieves the set of links that overlap with the query range and point to the queried categories and then, by following those links, it retrieves the actual objects.

More precisely, STITCH first probes the reference index and finds the cells overlapping the query volume. It reduces the search space instantly and only fetches the links to partitions as well as the partition's MBB from the grid cells for each of the categories $q \in Q$ queried. STITCH initially

Algorithm 11: STITCH Querying Algorithm

Input: L: reference index
g: # of uniform grid cells in the reference space
query: spatial range query
datasets: set of queried spatial data sets, each corresponding to a distinct category
Output: *objects*: result set of objects

```
// set of grid cells overlapping the query:
C ← cellNr(g, query);
foreach dataset D ∈ datasets do
    P ← ∅ ;                                   // set of qualifying links
    foreach c ∈ C do
        fetch from cell c of L the metadata_records of D;
    foreach m ∈ metadata_records do
        if m.MBB does not overlap query then
            discard m;
        else
            P = P ∪ m.link;
    foreach p ∈ P do
        retrieve the disk page pointed by p;
        foreach object o on the disk pages do
            if o overlaps query then
                objects = objects ∪ o;
return objects
```

purges any duplicate links retrieved from different grid cells that intersect with the same data partition. In a next step STITCH further reduces the number of category data pages needed to be retrieved by discarding the links associated with an MBB that does not overlap with the query volume. With the set of remaining links, STITCH retrieves the disk pages that contain the actual objects. In a last filtering step, STITCH discards the objects whose MBB does not overlap with the query volume. The pseudo code of the complete STITCH querying algorithm is described in Algorithm 11.

Clearly, STITCH avoids the repeated traversal of the index structure because we query the reference index only once. In addition, using a uniform grid as the reference index enables efficient query execution: STITCH can readily calculate the intersection between the query and the grid cells and obtain the offsets of those intersecting grid cells on disk by performing a hash lookup. Finally, STITCH avoids retrieving unnecessary data because it reduces the number of links effectively by first only retrieving links pointing to categories that are queried for and ultimately discarding the links associated with MBBs that do not overlap with the query range.

5.1.7 Experimental Evaluation

In the following section we describe the experimental setup & methodology and demonstrate the benefits of STITCH using real neuroscience workloads. We compare STITCH against the two strategies described before, `1-for-each` (one index per category) and `all-in-1` (a single index for all categories), and present a detailed breakdown of the performance. Lastly, we conduct a sensitivity analysis where we vary specific data set, workload and configuration parameters to better understand STITCH's behavior.

Experimental Setup

Hardware Configuration. The experiments are run on a Linux Ubuntu 12.04 machine equipped with 2x Intel Xeon Processors each with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 cache and 48GB RAM at 1333MHz. The storage consists of 2 SAS disks of 300GB capacity each.

Software Setup. We implemented all the approaches single-threaded in C++ and compiled with `g++` with the maximum optimization level. We set the disk page to 4KB for all the approaches and for both metadata and data files. All the indexes have the same memory footprint during the index building phase (1GB) and we use only one disk during the experiments (i.e., no RAID configuration). For all experiments we assume cold system caches so the OS caches and disk buffers are cleared (overwritten with an empty file) before each query is executed.

Competing Approaches. We experimentally compare STITCH against the following spatial indexes: FLAT `1-for-each`, FLAT `all-in-1` and GRID (`1-for-each`). We omit comparisons against the R-Tree because it is outperformed by FLAT [144]. For our workload, FLAT `1-for-each` answers queries by up to 2× faster compared to R-tree `1-for-each`, and FLAT `all-in-1` is ∼ 6× faster than R-tree `all-in-1`. We use the original implementation of FLAT [144] that the authors made available to us and our own implementation of a uniform disk-based grid index. Given that it is unrealistic to index an entire data set in-memory before flushing the data to disk, GRID inherently writes data for each grid cell to disk individually (and thus distributed) when the memory buffer becomes full, which can result in random reads during the querying phase. Similarly to STITCH, GRID maintains an in-memory hash table storing the disk offsets of all the non-empty grid cells. To avoid replicating objects that overlap with multiple grid cells, in our GRID implementation we adopted the following strategy proposed in [138]: each object is assigned only to the grid cell that encloses its center. During querying, to ensure that all the objects intersecting a grid cell are retrieved, the query range is enlarged by the width of the biggest object in each dimension.

Configuration. Given the absence of heuristics, we perform a parameter sweep to identify the best performing configuration for STITCH and GRID. For GRID, the configuration with 60^3 cells balances the number of objects retrieved and disk spatial locality. STITCH performs best with 100^3 cells for the reference index.

Experimental Methodology

Data Sets. We use data sets that model a small part of the brain with a surface mesh consisting of 3D triangles in a volume of 285 μm^3. Each neuron type forms one category. Different categories are stored in different data sets (similarly to the cosmology use case described in Section 5.1.2). In other words, each data set we use in our experiments (10 data sets in total) corresponds to a subset of the neurons that are contained in the same brain volume. We approximate the 3D triangles with axis aligned MBBs and store only these MBBs along with an object identifier in the object partitions. All approaches therefore only test for overlap between the stored MBBs and the query volume. The coordinates of the MBBs are represented using double precision floating point numbers while the object identifier is an integer. Each data set occupies $\leq$ 5 GB on disk, and in total the indexed data is $\sim$ 45 GB.

Benchmark. Lastly we define a benchmark which consecutively executes 200 spatial range queries of varying sizes. The size, aspect ratio and the location of each query is randomly chosen. The average query volume in the benchmark is $10^{-6}\%$ of the entire universe. This benchmark is derived from our neuroscience use-case where specific subvolumes are retrieved with range queries for analysis purposes. As different subregions in the brain vary significantly in volume, the size of the issued queries varies accordingly.

Comparative Analysis

We first perform a comparative analysis among all competing approaches. For each experiment, we execute the same 200 queries taken from our real neuroscience benchmark to compare the total query execution time and the total amount of data retrieved. Additionally, we compare the time to build the indexes and the storage space required for each approach. Lastly, we compare the cost of adding a new data set in each approach. In each experiment, we increase the number of data sets that are queried to show how each approach scales as scientists increase the number of data sets (thus the total amount of data that is queried increases as well).

Query Execution Time. Figure 5.5 shows the query execution time as we increase the number of data sets queried, when all 10 data sets have been indexed a priori. In the same figure we also plot the line which shows the optimal strategy based on FLAT, i.e., indexing precisely the data sets needed for each query combination (`queried-in-1`, see Section 5.1.2). STITCH executes the queries fastest because the majority of the time ($\sim$ 53%) is spent on useful work, i.e., retrieving objects, not on traversing index structures or retrieving metadata information as other index structures do.

FLAT on the other hand incurs a higher overhead for retrieving metadata information. As Figure 5.6 shows, querying the index can take up to $\sim$ 90% of the total time for FLAT `1-for-each`. The FLAT `all-in-1` strategy requires roughly the same time as the same number of objects are always retrieved irrespective of how many data sets are queried. In addition to the cost of

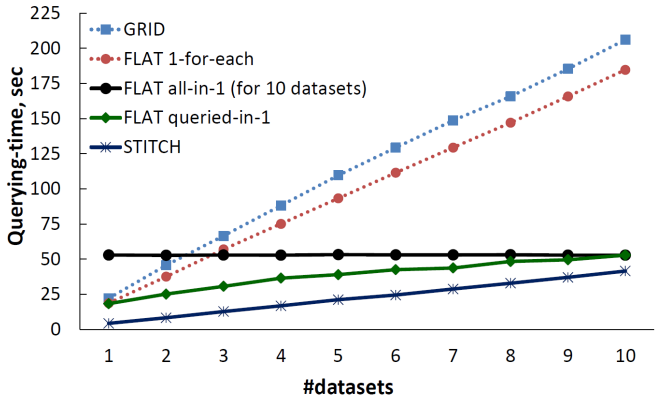


Figure 5.5 – Scaling-up with the number of queried data sets.

navigating the FLAT index, FLAT `all-in-1` retrieves all the objects within the query volumes from all data sets, as Figure 5.6 shows.

As a consequence, STITCH is 12.3× faster than FLAT `all-in-1` when only a single data set needs to be queried. When all data sets are queried, STITCH performs comparably to the `all-in-1` strategy, but is still 1.3× faster. Crucially, the trends in Figure 5.5 *do not imply a crossover point* where STITCH is outperformed by FLAT `all-in-1` if we add more data sets: the only reason why the query execution for FLAT `all-in-1` remains constant is because it always indexes 10 data sets throughout the experiment. If we add more data sets, the query execution time for FLAT `all-in-1` will increase (i.e., the flat black line corresponding to `all-in-1` will move higher up in the graph).

The `1-for-each` approach scales poorly compared to the other approaches. In particular, compared to STITCH, FLAT `1-for-each` performs ~ 4.5× slower on average. The key reason is that the cost of navigating each FLAT index grows due to having a separate index for each data set as shown in Figure 5.6. GRID `1-for-each` performs ~ 5× slower on average compared to STITCH. The primary reason is that the skew in the data sets results in grid cells that contain many objects and a lot of unnecessary data is retrieved during querying as Figure 5.6 indicates.

Data Retrieved. Comparing the measurements in the left side of Figure 5.6 with the respective approaches in the right side of the same figure, we can see that the data retrieved from disk correlates with the total query execution time and therefore is the most significant factor that defines the trend of performance for each approach (i.e., the execution time for all approaches is I/O bound). Operations such as testing overlap between partitions and querying MBBs - as well as filtering objects in case of the `all-in-1` approach - are performed while the data resides in memory and therefore do not significantly affect query execution. Although GRID retrieves by far the largest amount of data compared to all the other approaches, the query execution time is not affected severely because the access pattern is mainly sequential.

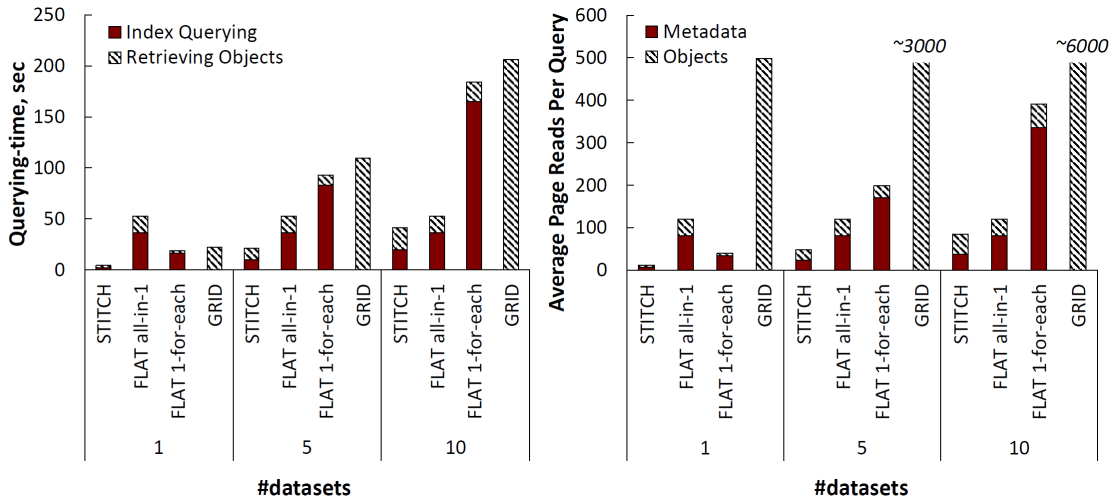


Figure 5.6 – Query execution time breakdown (left) and breakdown for the pages read per query (the page size is 4KB) (right).

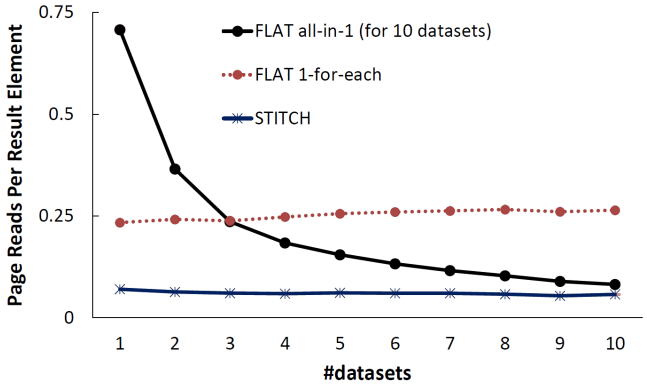


Figure 5.7 – Total number of page reads per result element.

To further study the different index structures and quantify their overheads, we measure the number of page reads per result element as we increase the number of data sets, focusing on FLAT and STITCH. Figure 5.7 shows that STITCH has a fixed overhead irrespective of the number of data sets. FLAT `all-in-1` has a big overhead when only 1 data set is queried out of the 10 indexed data sets, but it converges to STITCH's performance when all 10 data sets are queried. Finally, FLAT `1-for-each` has a higher overhead than STITCH which is slightly increasing with more data sets.

Index Time. Considering index building, the `all-in-1` approach is the most time-consuming. FLAT index building requires (externally) sorting the data on each dimension to create partitions, building a tree on top of the partitions and then using the tree to find neighboring partitions. Building many smaller indexes is more efficient, mostly because each index operates on a smaller

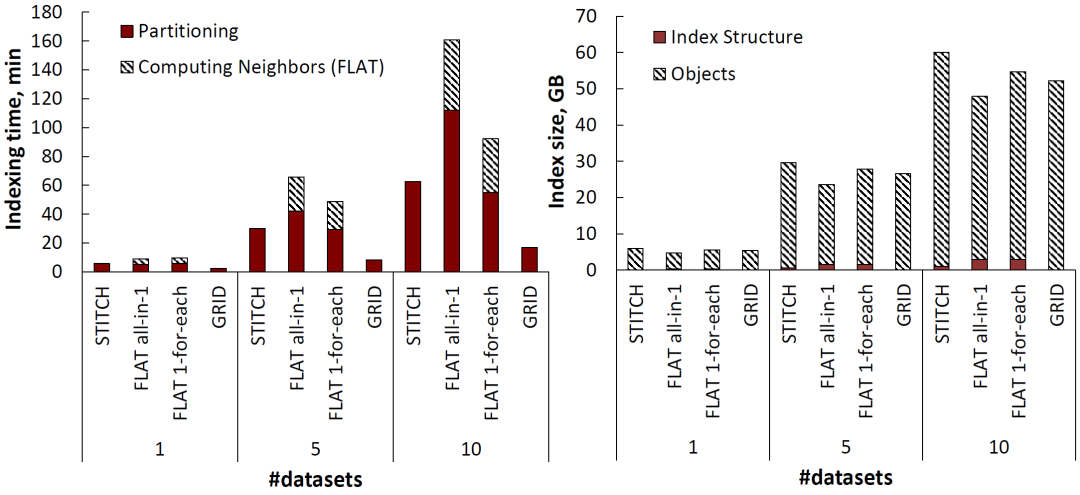


Figure 5.8 – Overall time to index (left) and Index size (right).

data set and thus requires fewer (or even zero) external passes for sorting. In our experiments, `1-for-each` outperforms `all-in-1` by ~ 40% as shown in Figure 5.8. STITCH indexes faster than FLAT, because virtually no time is required for creating a grid index on the reference space and constant time is required for computing the overlap between the grid cells and the data partitions. Therefore STITCH spends only 1% of the time for linking the reference index and the data sets while the remaining 99% is spent on partitioning the data sets. GRID indexes the fastest because it simply partitions the data uniformly and does not require an external sort.

Index Size. If we compare the storage space needed by all indexes in each approach, we see that the majority of the space is taken by the objects themselves (partitioned data sets), roughly 45 GB for all 10 data sets. STITCH requires the least amount of space (~1 GB) to store the metadata information. Both FLAT `all-in-1` and `1-for-each` have virtually the same index structure and therefore require a similar amount of space - around 3GB for 10 data sets while GRID does not need to store any index structure at all. In terms of the space needed to store the objects, both GRID and mainly STITCH introduce some empty space in the object pages, requiring 16% and 31% more space compared to FLAT respectively. In the case of STITCH, this extra space enables more fine-grained filtering and thus results in faster overall query execution.

Index Update. Scientists often acquire more data of the phenomena being studied. Providing indexing support for newly added data sets is therefore important. We thus analyze the cost of adding a new data set in each approach. We initially index 9 out of our 10 data sets, and measure the time for adding the 10th data set (of size 4.9 GB). FLAT is designed for bulkloading and therefore it is more efficient to re-build `all-in-1` from scratch. `1-for-each` only requires to build the index for the new data set and is therefore much cheaper as shown in Figure 5.9. STITCH needs to partition the new data set and link the partitions to the reference index which is a faster process than building the whole index in `1-for-each`. Overall STITCH finishes the

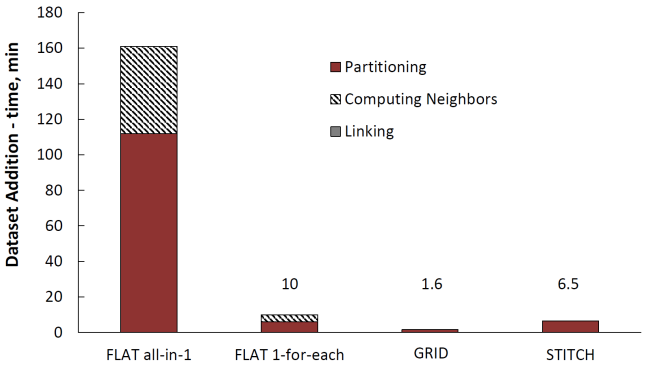


Figure 5.9 – Extending an existing index with a new data set.

update about 35% faster than `1-for-each`. GRID is the fastest approach as it simply partitions the new data set uniformly without the need to sort it first.

Sensitivity Analysis

In the following, we perform an analysis of STITCH to understand the impact of different workload and configuration characteristics on the performance. The following experiments are performed using the same query workload described in Section 5.1.7 and indexing 4 neuroscience data sets while querying all 4 of them.

Scaling with Data Set Size. In this experiment we execute the same queries on data sets with increasing size and we study the impact on the number of links between the reference space and the category partitions. We define three cases: (1) we index and query successively four small sized data sets (a total size of 9 GB), each one containing a small set of neurons, (2) we use four data sets of medium size (a total size of 18 GB) and (3) we query four big data sets (a total size of 40 GB) containing a large set of different types of neurons. For a fair comparison, the granularity of the reference index is not adjusted to the size of the data sets (100^3 cells are used as in the previous experiments). As all the data sets are contained within the same reference space, adding more neurons results in increasingly denser data sets. The overlap between the grid cells and the category partitions thus increases. As shown in Figure 5.10 (left), for a given grid cell, there are more links to category partitions outside the query range. The number of replicated links (links retrieved from multiple grid cells) and the number of links that point to the actually overlapping category partitions (hits) increase as well (because the query size is constant). The right side of Figure 5.10 shows that as the data set size increases, there are more objects in the query result, and retrieving them becomes the dominant factor.

Grid Resolution. The number of grid cells has an impact on the number of links that need to be retrieved from disk to evaluate a query. When the grid resolution is too low, the grid cannot

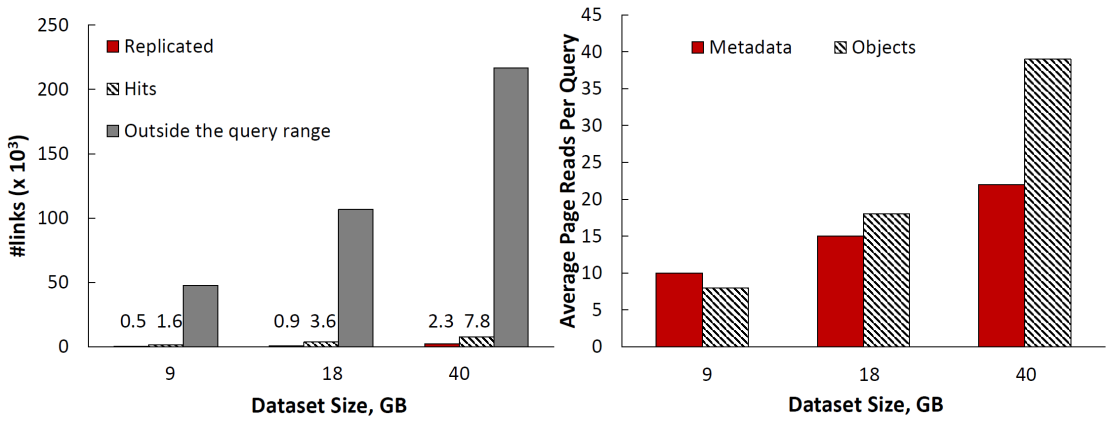


Figure 5.10 – Number of links (left) and Pages read per query (right) for increasing data set sizes.

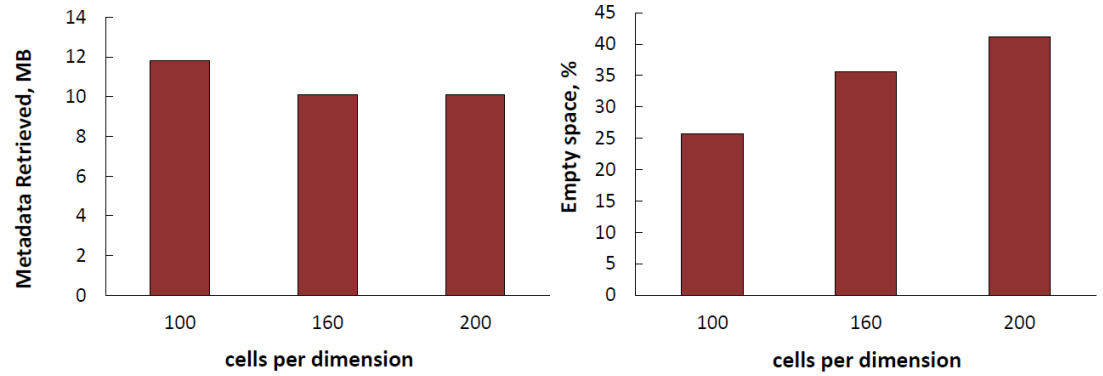


Figure 5.11 – Amount of retrieved metadata (left) and percentage of empty space in objects pages (right) for increasing grid resolution.

effectively prune the links to the category partitions that fall outside the query volume. Figure 5.11 shows that the amount of data retrieved from the reference index (left-hand side) decreases as the resolution increases from 100 to 160 cells per dimension. However, increasing the resolution further does not help in reducing the amount of retrieved data. The right-hand side of Figure 5.11 shows the impact of the grid resolution on the category partitions. As the resolution increases, `Sliced-DOP` leaves more empty space in the object pages (i.e., creates category partitions with a smaller number of objects), resulting in an overhead of up to 40% for 200 cells per dimension.

Query Volume. In this experiment, we increase the volume of the queries from 10^{-6}% to 10^{-4}% of the entire universe volume. Since the grid resolution is fixed, when bigger queries are executed, an increasing number of grid cells overlap the query. However, not all the links inside those cells are pointing to a category partition that overlaps the query. As Figure 5.12 shows, the increase in

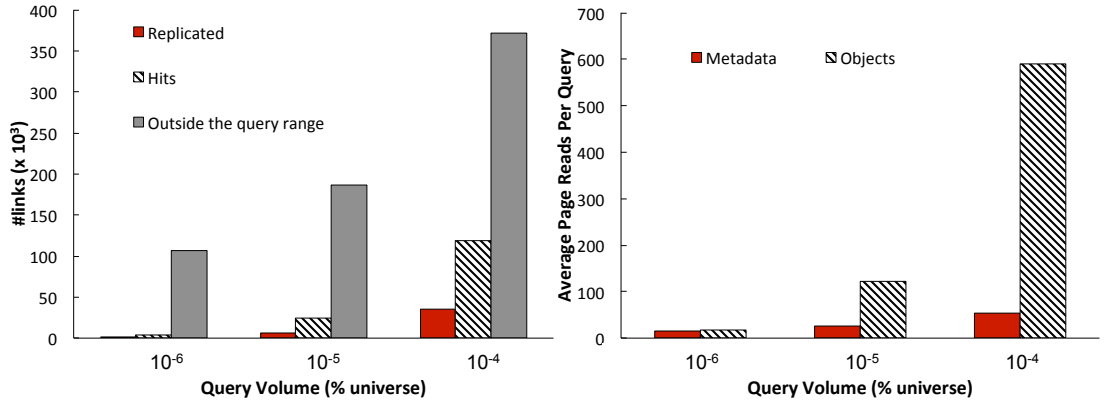


Figure 5.12 – Number of links (left) and Pages read per query (right) for increasing query volume.

the number of overlapping cells results in an increasing number of links that do not overlap the query volume. The same figure (right) shows that as the query volume (and the number of cells that overlap with the query) increases, more pages are read for both metadata and objects, but retrieving objects is the dominant factor.

5.1.8 Discussion

Using space-oriented partitioning in the reference space, we cover the entire universe of all categories without having a priori knowledge of the data distribution, and thus ensure that all category partitions (even those of future categories) are "stitched" to at least one partition in the reference space. In particular, STITCH employs a uniform grid which provides fast access to the query region and allows to link new categories efficiently. However, the grid resolution has to be defined statically. Alternatively, other space-partitioning indexes could be used as the reference index, such as octrees [80] or kd-trees [18].

5.2 Part II: Incremental Indexing for Multiple Spatial Data Sets

5.2.1 Introduction

In astronomy, biology, neuroscience and other disciplines, scientists are increasingly overwhelmed by the amount of data they have at their disposal. With advances in sensor technology and supercomputing for large-scale simulations, the amounts of data scientists have to analyze grow rapidly. Today's tools are frequently inadequate to analyze this large data and answer key questions: executing even simple queries (such as spatial range queries) is time-consuming. Data sets can, of course, be indexed a priori to accelerate access, but the areas analyzed are rarely known beforehand and also only represent a small subset of the entire data set, making upfront indexing an undue overhead.

In neuroscience, for example, scientists need to explore multiple massive data sets originating from different sources [106] to investigate particular areas of the human brain. The data in this use case is spatial and originates from different instruments (e.g., patch clamp, brightfield spectroscopy, MRI) of different resolutions. To perform an analysis, they need to query small parts of different combinations of data sets, each of a size in the order of terabytes. What areas of the data sets they need to access and what combinations of them are not known a priori. It is consequently unclear what parts of what data sets need to be indexed. It is however clear that fully indexing all data sets introduces considerable overhead which is unlikely to pay off.

More formally, the problem is the efficient exploratory analysis of multiple spatial data sets through the execution of spatial range queries: given n data sets and a $m-$sized subset of data sets ($m \leq n$), scientists need to efficiently execute a spatial range query q on each of the m data sets. What combinations of m data sets will be queried together and what spatial ranges q will be accessed is not known beforehand. The challenges thus are twofold (a) what areas in the data sets are accessed and (b) what data sets are accessed together.

Multiple spatial indexes have been developed to accelerate access to spatial data sets addressing the first challenge [54]. All of them, however, require the entire data set to be indexed at once. Incremental approaches to indexing (or reorganising data layout) have been developed for relational data stored in main memory [75, 76], but not for spatial data on disk. To the best of our knowledge, incrementally indexing spatial data on disk and accelerating access to multiple data sets queried together, are two challenges that remain unaddressed in literature

Space Odyssey, the approach we develop, addresses both challenges and enables the efficient exploration of multiple spatial data sets. While the spatial data sets are being queried, Space Odyssey indexes them incrementally to accelerate access to the data sets in general and to the frequently queried areas in particular. At the same time, Space Odyssey reorganises the data layout on disk so that parts of the data sets queried together can be retrieved more efficiently. By incrementally indexing and reorganising the data, Space Odyssey accelerates explorative analysis of spatial data, substantially reducing query-to-insight time: it answers up to several hundred queries (more

than half the queries of our benchmark, see 5.2.6) by the time the fastest existing approach has merely indexed the data.

5.2.2 Space Odyssey Overview

Space Odyssey enables exploratory access to multiple spatial data sets without pre-processing the data. Instead, Space Odyssey uses incoming queries as hints for reorganizing the physical data layout to better serve queries.

First, to enable efficient access to precisely the areas queried in individual spatial data sets, Space Odyssey indexes the data sets incrementally. Second, to better support querying the same areas in different data sets, it adapts the physical layout on disk, storing together the areas that are queried together to accelerate retrieval.

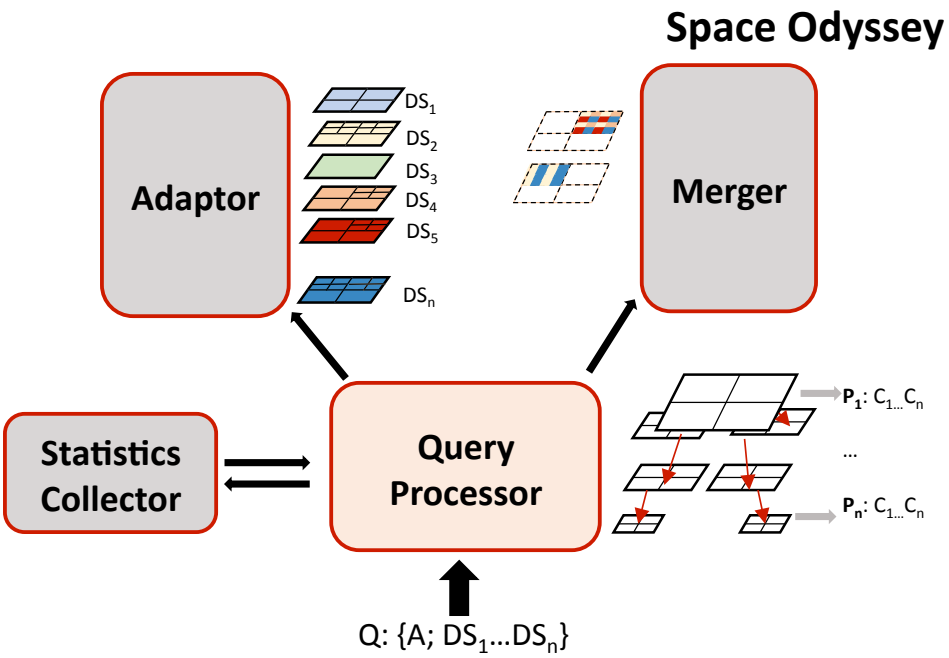


Figure 5.13 – Space Odyssey: components, data structures and a snapshot of the physical layout.

Figure 5.13 illustrates the architecture of Space Odyssey – its components, data structures and a snapshot of the physical layout. The Adaptor is responsible for the incremental indexing and the Merger performs operations related to the physical layout. Finally, the Query Processor orchestrates the overall query execution process using information provided by the Statistics Collector.

5.2.3 Incremental Indexing

Indexing all data sets a priori has the major drawbacks that (a) scientists must wait until all data is indexed before they can start querying and (b) data that is never queried is indexed in a time-consuming process.

Space Odyssey therefore uses incremental indexing where in every step (with every query) it further refines the index structure in the frequently queried "hot" areas to accelerate future queries. To keep the overhead of incremental indexing low, it employs space-oriented partitioning, which has a lower processing overhead compared to data-oriented partitioning [54].

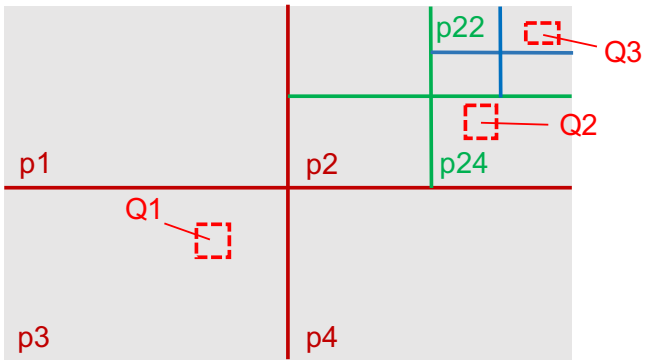


Figure 5.14 – Incremental indexing strategy (in 2D).

Data Structure. More precisely, Space Odyssey incrementally builds an Octree [54] on each data set queried. The Octree is the index of choice since we want to introduce minimal overhead during the query execution and thus, we split each dimension to a minimal number of partitions which corresponds to 2^d partitions in a d-dimensional space. Figure 5.14 illustrates the indexing process with $d = 2$, i.e., 4 partitions per level. The indexing process starts with the first query $Q1$ where Space Odyssey partitions the space uniformly into four partitions (p1, p2, p3, and p4). It scans the data set and assigns each object to the partitions it overlaps with. When the second query arrives ($Q2$), Space Odyssey identifies the partitions that it intersects with (only p2 in our example), refines these partitions, i.e., divides p2 into four sub-partitions (p21, p22, p23, and p24), and reassigns their objects to the new partitions. In the same process it checks whether the objects of the qualifying new partitions are inside $Q2$. Space Odyssey applies the same procedure for query Q3 and all subsequent queries.

Space Odyssey refines partitions to curb the amount of data retrieved and checked for intersection with the query. Intuitively, for efficient performance, we want the partition size to approximate the query size. Then, in the best case, a query hits only one partition which covers just the queried range so that a single sequential scan of the partition retrieves all required objects. In the worst case, the query intersects 2^d partitions. Refining a partition further than approximately the query size only incurs unnecessary processing overhead (for the actual refining as well as retrieving and scanning multiple resulting partitions). Therefore, to control the degree of

refinement, Space Odyssey uses a *refinement threshold* (*rt*). A partition is refined following the execution of a query if $\frac{V_p}{V_q} > rt$ where V_p and V_q are partition and query volumes, respectively.

The incremental refinement strategy spreads the overhead of building the index and reorganizing the data on disk over several queries. Areas frequently queried will be indexed fully at a fine granularity, such that range queries in these areas are executed as efficiently as if executed on a fully built Octree. Areas untouched by past queries will be partitioned at a coarser granularity thus future queries in these areas can also benefit from the adaptive partitioning.

Optimizations. In case the query size is significantly smaller than the size of the partition it hits, a considerable number of queries may be executed before the partition is refined sufficiently. The number of queries that need to hit a partition before it reaches the finest level of refinement determined by the refinement threshold are given by the following equation:

$$\log_{ppl}(V_p/(V_q \times rt))$$

where *ppl* is the number of partitions per level and $ppl = 2^d$ in a standard Octree. To converge faster, a bigger *ppl* can be used.

Since we use space-oriented partitioning, a spatial object can intersect with several partitions which introduces additional intersection tests. To avoid object replication and eliminate unnecessary tests while curbing the memory footprint, we translate the problem of indexing volumetric objects to indexing point data using the query window extension technique [138]. Space Odyssey assigns each object *o* to a partition based on its center and keeps track of the maximum object extent (*maxExtent*) in each dimension. Then, to answer a query correctly ensuring that all intersecting objects are retrieved, the query range is extended by *maxExtent* and all the cells the extended query overlaps with are inspected.

Finally, Space Odyssey performs the updates in-place, i.e., it reads a partition *p*, refines it and reuses the disk pages where partition *p* was stored for the newly created partitions. If there is not enough space in the pages initially storing *p*, new disk pages are allocated and appended at the end of the file.

5.2.4 Incremental Merging

By building data structures incrementally we can significantly decrease the data-to-query time. At the same time we have the opportunity to optimize the placement of data structures on disk to accelerate the queries executed.

Particularly in the case where multiple data sets are analyzed, apart from building an index structure incrementally for each data set, Space Odyssey also rearranges the data on disk to store together the areas in different data sets that are frequently queried together. Doing so allows

Space Odyssey to avoid random disk accesses for retrieving the same area in different files and thereby accelerate queries.

Merging Partitions. While executing queries Space Odyssey keeps statistics about the data sets queried together and the partitions retrieved from them. More precisely, given queries of the form $Q = \{A; C\}$ where A is the area queried and $C \subset \{DS_1, \ldots, DS_n\}$, it will store: 1) how often a given combination C is accessed and 2) what partitions are retrieved from C, i.e., what partitions $p \in P$ overlap with A.

Once the number of retrievals for a particular combination C of data sets exceeds a preset *merging threshold* (*mt*), Space Odyssey merges the data for all the partitions $p \in P$ retrieved in the context of C. It iterates over all partitions that have been queried for in C, retrieves them from every data set $DS_i \in C$ $(1 \leq i \leq n)$ and merges them on disk. Note that some of the merged partitions may be retrieved less frequently by past queries than others, but the overhead of including them in the merged file is minimal while there is a benefit in case they are accessed more frequently in the future.

Lastly, Space Odyssey merges data only for combinations of size $|C| \geq 3$ because merging is more beneficial for bigger combinations as it prevents (random) accesses to a large number of data sets.

Physical Layout. Space Odyssey creates a new *merge file* where it stores the partitions P from different data sets queried together in a combination C so they can be read sequentially and hence more efficiently once they are again queried together. The partitions in the merge file are copies, meaning that Space Odyssey also keeps the original partitions to support efficient querying on an individual data set DS.

For a given partition $p \in P$, the merge file physically stores the objects contained in p from each data set $DS_i \in C$ $(1 \leq i \leq n)$ sequentially. Given for example data sets DS_x, DS_y, DS_z $(1 \leq x < y < z \leq n)$, Space Odyssey stores objects from DS_x on the first disk pages, followed by objects from DS_y followed by DS_z. Doing so allows to retrieve efficiently only the objects belonging to a queried subset of all data sets merged (e.g. DS_x and DS_z) by reading them sequentially while skipping over the rest (DS_y). The merge file is append-only, i.e. new partitions are always added at the end of the file.

Space Odyssey incrementally builds index structures per data set and the same regions in different data sets may thus have a different level of refinement. In data set DS_x, for example, the area may still only be covered by one partition p while it is divided into eight partitions in DS_y. Including copies of the unrefined partition p in merge files adds the challenge of having to refine all the copies once refinement of p is triggered by a new query, thereby introducing substantial overhead. Space Odyssey addresses this issue by only merging partitions which are at the same level of refinement. Additionally, in our current implementation the merged partitions are not refined any further.

Managing Storage Space. Space Odyssey maintains a space budget for merge files (and thus replicated partitions). Once the space budget is exceeded it removes the least recently used merge files to adhere to the budget.

Open Issues. Building a merged index for the hot areas where multiple data sets are queried together significantly accelerates queries but several challenges need to be addressed to fully automate the merging and maximize the performance gains. In particular, we plan to develop a cost model which indicates how to adapt the parameters (minimum size of combination to be merged $|C|$ and mt) at runtime based on the workload. Additionally, we plan to investigate the benefits of merging partitions at different refinement levels and examine alternative strategies for doing so, e.g., should all partitions be refined to the same level as the finest partition before merging or as the coarsest, or shall we allow multiple refinement levels to coexist in the merged index. Lastly, we plan to improve disk space management to avoid the replication of a data set which is used in several different combinations whenever possible.

5.2.5 Space Odyssey Query Execution

To efficiently execute queries and take advantage of merge files, i.e., to decide whether to retrieve areas from individual data sets DS_i ($1 \leq i \leq n$) or from merge files, Space Odyssey maintains a directory where it keeps information about what partitions of what combinations of data sets are stored together.

Once a query $Q = \{A; C_q\}$ ($C_q \subset \{DS_1, \ldots, DS_n\}$) is issued, Space Odyssey checks what partitions intersect with A and whether these partitions are stored in a *merge file*. There are four possibilities:

1. *Exact merge file:* if the exact combination C_q is stored in a merge file and contains the partitions intersecting with A, then it is used to retrieve those partitions sequentially.

2. *Superset:* if a superset $C \supset C_q$ is stored, i.e. the merge file contains more data sets than the ones requested, then the merge file will still be used. Using the merge file is more efficient than accessing individual data sets thanks to the internal organization of merge files: the objects from each data set are organized sequentially, meaning that they can be read efficiently but also that if data from a particular data set is not needed it can be skipped.

3. *Subset:* if a subset $C \subset C_q$ is stored, i.e. the merge file contains fewer data sets than the ones requested, then Space Odyssey uses the merge file to retrieve all data from the subset C as well as other merge files or individual files to retrieve the remaining data sets $C_q \setminus C$. The decision which of the merge files to use is based on maximizing the number of data sets already stored in a merge file and thus minimize (random) access to individual files.

4. *No merge file:* if no merge file exists for a combination C, individual files are used.

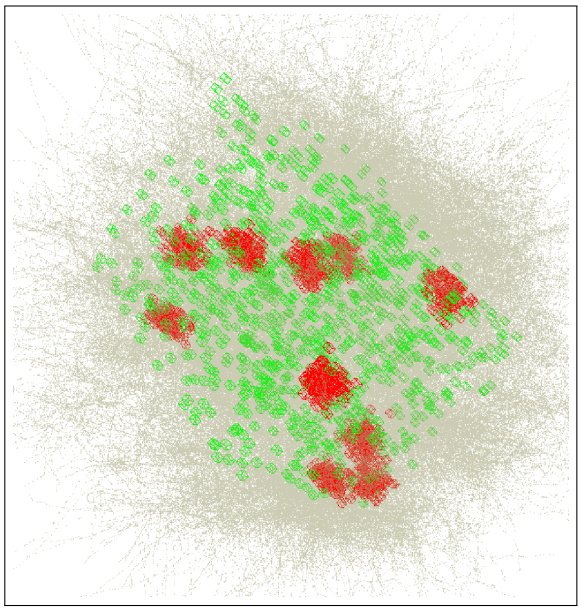


Figure 5.15 – Clustered (red) and uniform (green) range queries on one neuroscience data set (grey).

5.2.6 Experimental Evaluation

In this section we first describe the experimental setup and methodology and then demonstrate the behavior of Space Odyssey by comparing it against state-of-the-art spatial indexing approaches using real neuroscience data sets.

Experimental Setup

Hardware Configuration. The experiments are run on a Linux Ubuntu 12.04 machine equipped with 2x Intel Xeon Processors each with 6 cores running at 2.8GHz, with 64kb L1, 256KB L2 and 12MB L3 cache and 48GB RAM at 1333MHz. The storage consists of 2 SAS disks of 300GB capacity each.

Competing Approaches. We have implemented Space Odyssey and set its configuration parameters $rt = 4$, $ppl = 64$, and $mt = 2$. Also, we consider the following two approaches:

FLAT: the state-of-the-art indexing technique for spatial range queries for which we obtained the source code from the authors [144]. FLAT relies on data-oriented partitioning, more specifically the Sort Tile Recursive algorithm (STR [95]). As we need to index multiple spatial data sets, we implemented two strategies: `1-for-each` (1fE) and `all-in-1` (Ain1). The first strategy, 1fE, builds one index for each data set. To perform a query, all the indexes corresponding to the queried data sets are probed and the union of the retrieved results forms the final answer. The second strategy, Ain1, builds only one index structure containing all

the spatial objects from all the data sets. To perform a query, the index is probed and items belonging to data sets which are not queried are filtered.

Grid: a static, uniform grid-based technique where the indexed space is uniformly partitioned into a fixed number of cells (space-oriented partitioning). We use our own implementation. The objects are assigned to the grid cells in-memory and flushed to disk when the memory buffer becomes full. Similarly to Space Odyssey, replicating objects to multiple grid cells is avoided by using the query window extension technique [138]. The configuration is set to 60^3 cells, which we determine through a parameter sweep, given the absence of heuristics. When indexing with a Grid, the partitioning is fixed and does not depend on the data so there is essentially no distinction between the two strategies (`1-for-each` and `all-in-1`). Therefore, we only implemented the `1-for-each` (1fE) strategy.

Software Setup. All implementations are written in C++, they are single-threaded and compiled using `g++` (v4.9.2) with the `-O3` optimization flag. The disk page size is set to 4KB. To obtain realistic runtimes, and simulate a scenario where data set sizes are significantly larger than the main memory size, all techniques are restricted to have the same main memory footprint (1GB). For all experiments only one disk is used (i.e., no RAID configuration) while the OS caches and disk buffers are cleared (overwritten with an empty file) before each query is executed (i.e., to avoid caching effects).

Data Sets. We use 10 real neuroscience data sets that we obtained from our collaboration with neuroscientists in the Human Brain Project [106]. Each data set represents a subset of neurons contained in the same brain volume. The neurons are modeled with a 3D surface mesh. Figure 5.15 shows a 2D projection of one brain area. An identifier is attached to each object to distinguish items belonging to different data sets. Each data set requires approximately 5 GBs of storage on disk, resulting in $\sim$ 50 GBs of data in total.

Queries. Based on the previously described use cases, we synthetically generate queries each having a fixed volume (*qvol*) of 10^{-4}% of the total brain volume. We use a *clustered* distribution and choose a number of *clustercenters* ($|clusterscenters| = 10$). Query centers are distributed around the cluster centers following the Gaussian distribution ($\mu = 0$, $\sigma = qvol \times 10$). For completeness and to test non-skewed cases, we also generate uniformly distributed query centers. Figure 5.15 illustrates the query ranges of both distributions.

To determine the subset of data sets that are queried for each spatial range, we use a synthetic distribution generator based on Gray et al. [64]. The distributions we use are: (1) heavy hitter, (2) self-similar, (3) Zipf, and (4) uniform. These distributions have been used in other studies for similar purposes (e.g.,, in [37, 136]). In the heavy hitter distribution, one combination of queried data sets accounts for 50% of all possible combinations, while the other queried combinations are chosen uniformly from the remaining ones. The self-similar distribution uses an 80–20 proportion, and the Zipf distribution uses an exponent of 2. For the non-skewed scenarios where

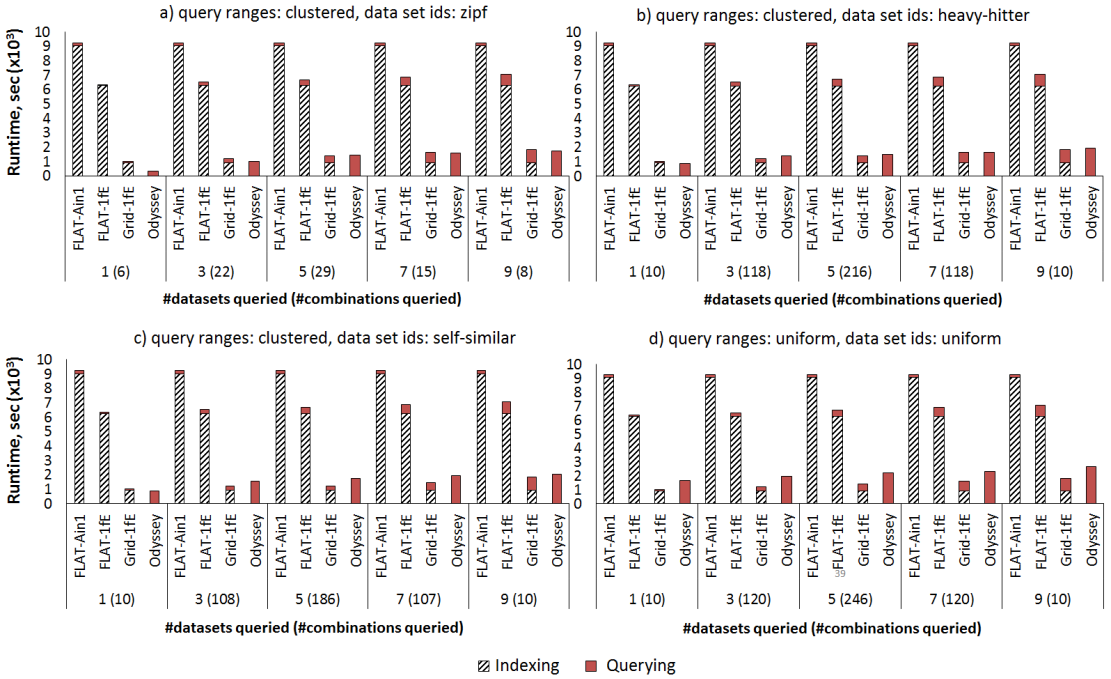


Figure 5.16 – Performance when varying the number of queried data sets for each distribution.

the centers of the spatial ranges are uniformly distributed in space, the combinations of data sets queried together are also chosen randomly from a uniform distribution.

Experimental Analysis

Total Processing Cost. Figure 5.16 depicts the total workload processing time when the number of queried data sets increases from 1 to 9. The $x-$axis in the graphs also shows the number of different combinations that are queried. Note that the combinations actually queried are often fewer than the total number of possible combinations and depend on the distribution. For Space Odyssey's competitors, the processing time is additionally broken down into indexing and querying. For Figures 5.16a, b, and c, we fix the query range distribution to clustered. For Figure 5.16d we uniformly choose both the query ranges as well as the queried data sets in order to demonstrate the worst-case performance where neither hot areas nor popular combinations exist.

We make the following observations. First, building FLAT takes at least 2 times longer than processing the entire workload of 1000 queries with Space Odyssey. Indexing with FLAT is up to ×5 slower comparing to the simple uniform Grid[3]. As such, only Grid is competitive in terms

[3]Favoring Grid, we assume that the optimal configuration is known. Otherwise, several builds of Grid are required to tune it.

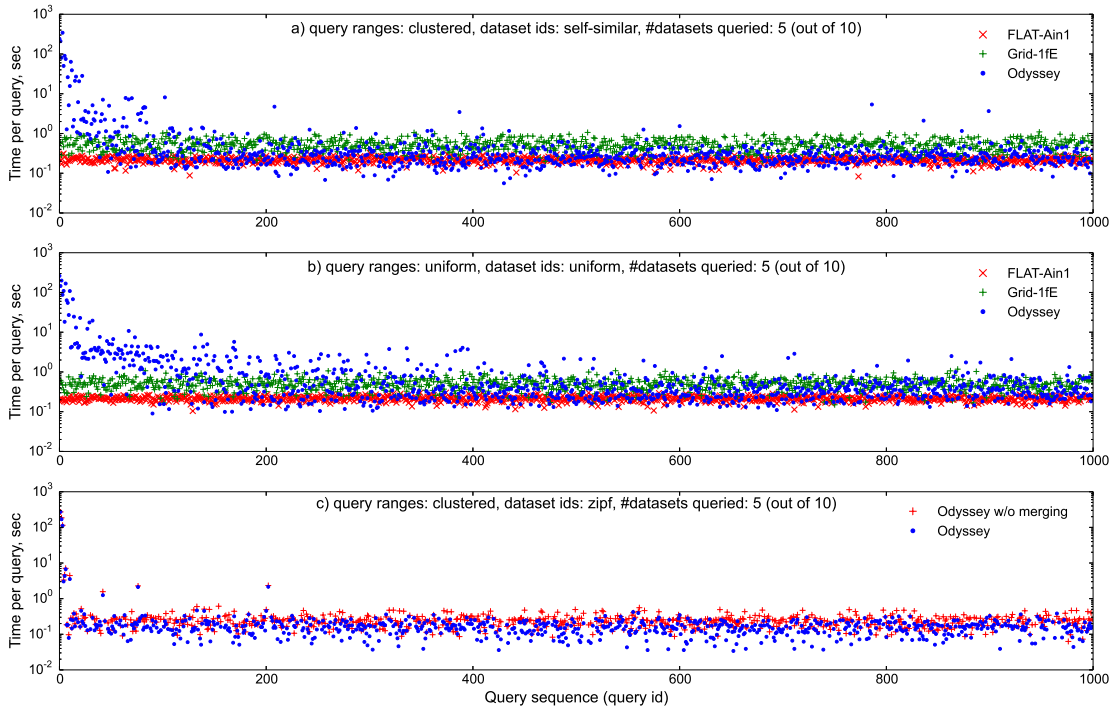


Figure 5.17 – Query times for each query in a sequence.

of overall data-to-query time when compared to Space Odyssey. Nevertheless, by the time Grid finishes indexing the data, Space Odyssey has already answered half of the queries on average.

Second, once the related approaches have indexed the data, they may process individual queries faster than Space Odyssey. While FLAT is the slowest to build, its variants report the fastest querying times compared to other approaches – up to ×6, and ×9 faster than Grid, and Space Odyssey, respectively (considering just the querying time for static approaches and the total time for Space Odyssey). The important aspect of Space Odyssey however is that it has the lowest data-to-query time, because there is no need to build complete indexes for all the data sets in advance.

Third, Space Odyssey is a middle ground between the `1-for-each` and the `all-in-1` strategies. The `1-for-each` (1fE) strategy accesses individual (smaller) indexes and only for the data sets queried. Consequently, the query processing cost increases with the number of queried data sets. The `all-in-1` (Ain1) strategy, on the other hand, always operates on a huge index structure and suffers from unnecessary data accesses. As such, when the number of queried data sets is less than 5, 1fE is preferred over Ain1. Space Odyssey follows a hybrid strategy, where the individual data sets are indexed adaptively (similarly to 1fE) but hot areas from different data sets are merged together (similarly to Ain1).

Finally, while all related approaches are insensitive to skew in the workload, the adaptive mechanisms in Space Odyssey are able to exploit it. For example, in Figure 5.16, when the queried data set combinations are coming from the very skewed zipf (a) and heavy-hitter (b)

distributions, Space Odyssey quickly refines the hot areas, merges the partitions of the popular data sets together, and is often able to perform most of the queries before even Grid finishes building. This is not the case with the less skewed self-similar distributions (Figure 5.16c), where Grid (once its building phase is over) answers individual queries faster than Space Odyssey most of the time. When both query ranges and queried data sets are uniformly distributed (Figure 5.16d), Space Odyssey cannot benefit from adaptive refinement and thus takes longer than Grid to process the entire workload of 1000 queries.

Query Performance. In Figure 5.17 we show the response time for each query in a sequence when 5 data sets are queried. In Figure 5.17a the queries are clustered and the queried combinations are chosen from the self-similar distribution while in Figure 5.17b both the queries and the combinations are chosen from a uniform distribution. We study Space Odyssey and two approaches that were previously identified as the most competitive ones (in terms of querying performance): FLAT-Ain1 and Grid-1fE. In both cases, the very first query is the most expensive for Space Odyssey as it fully scans and partitions at the first (coarsest) level the raw data files for all 5 data sets in the combination. Nevertheless, we observe that Space Odyssey converges to the speed of the fully indexed case under both skewed (Figure 5.17a) and uniform (Figure 5.17b) scenarios. As expected, however, the convergence is slower in the uniform scenario. FLAT-Ain1 has consistently better and more robust performance than Grid-1fE because it is less sensitive to data skew. Once Space Odyssey has converged, its querying performance is between FLAT-Ain1 and Grid-1fE, while it performs some queries even faster than FLAT-Ain1. Finally, when an area that has not been previously refined and/or merged is queried, the querying time for Space Odyssey is still higher.

Effect of Merging. Lastly, to isolate the effect of merging partitions that are often queried together, we run Space Odyssey with and without merging enabled. In this experiment, clustered queries are produced using 5 instead of 10 *clustercenters* to increase the likelihood that the queries will benefit from merging. In Figure 5.17c, we plot the times only for the queries that request the most popular combination (for the zipf distribution, this combination is queried 751 times). While completely diffrent ranges (e.g., in different clusters) may be requested for the same combinations, we see that eventually Space Odyssey benefits from the merged partitions for the majority of queries. We observe 25% performance gain on average for queries accessing the merged partitions.

5.2.7 Related Work

Several approaches have been developed in recent years to adapt the data layout in response to incoming queries. To accelerate data access, database cracking [75, 76] iteratively refines the physical data layout in memory with each query, essentially amortizing the cost of index building over query processing. Similarly, incremental indexing strategies [59, 60] choose the indexes and create them as a side-effect of query processing. A user neither configures or creates indexes nor does she provide a representative workload. Instead, based on the queries executed, an adaptive

index is only partially materialized and optimized such as to fit the current workload and storage budget. As queries arrive, the index is adapted on the physical level to suit the workload. Known adaptive indexing techniques, however, require all data to be loaded upfront. Finally, several approaches skip pre-processing to reduce the cost of raw data querying. NoDB [7] accesses CSV data in situ to adaptively build positional and binary caches as a side-effect of query execution. RAW [86] extends NoDB and adapts its access layer using code generation techniques.

Numerous approaches have been developed to index spatial data [54]. Almost all spatial indexes, however, require the entire data set to be loaded upfront and do not adapt to the query workload. One representative exception are adaptive index structures [142] which rearrange the nodes of data-oriented hierarchical indexes (including the spatial R-Tree [54]) in response to queries so that they can be accessed sequentially on disk. However, this reorganisation is performed only after the index has been fully built. More recently, a query-aware incremental index has been proposed [124] for main memory workloads.

5.3 Chapter Summary

In this chapter we identify the challenge of efficiently exploring multiple spatial data sets with the same range query—a common task across scientific applications. Not knowing a priori which data sets will be queried makes it particularly challenging to accelerate access: indexing all possible combinations of data sets is not feasible as it takes too long and requires too much space leaving us to choose between two extremes – indexing each data set individually and using one index for all data sets. As we show in the first part of this chapter, neither of the two extremes is efficient: the first does not scale well with an increasing number of queried data sets, and the second is inefficient when only a small subset of the indexed data sets is queried.

Based on these key insights we develop STITCH, a novel disk-based index which combines data-oriented partitioning with space-oriented indexing. By using data-oriented partitioning for the data sets we can effectively address skew in the distribution of spatial objects. At the same time we refrain from using a hierarchical structure to access the partitioned data sets (and thus avoid the associated overhead) and instead link the data set partitions to a central space-oriented index. In particular, STITCH employs a uniform grid for its efficiency in building, querying and updating with new categories. Links are stored for all intersecting pairs of grid cells with data set partitions. Key to the approach is the use of Sliced Data-Oriented Partitioning (`Sliced-DOP`): to avoid storing and ultimately following an excessive number of links, the uniform grid guides the partitioning of the data sets. Our extensive experimental analysis shows that with these measures our approach outperforms the state-of-the-art by up to a factor of 12.3 for a real neuroscience workload.

In the second part of this chapter, we show that state-of-the-art methods require to index *all* data a priori including the parts never analyzed. As a consequence we develop Space Odyssey, an approach which incrementally indexes the bits of the data needed and that adapts the physical

layout of the data on disk to efficiently support the queries executed. This novel approach to incrementally indexing and reorganizing spatial data on disk shows benefits in decreasing the data-to-insight time.

6 Quadtree-based Bitmap Compression for Scalable Time Series Exploration

An increasing number of applications from finance, meteorology, science and others are producing time series as output. Analyzing the vast amount of time series is key to understanding the phenomena studied, particularly in the simulation sciences, where the analysis of time series resulting from simulation enables scientists to refine the simulated models. The challenge we are addressing in this chapter[1] is to efficiently find time series where the observed value exceeds or falls below a threshold during a given time interval. The desired threshold is not known a priori and therefore scientists perform an exploratory analysis where the result of each threshold query drives the formulation of the subsequent one. Existing time series access methods use single-attribute indexes, that index either the time or the observation value domain. On the other hand, threshold queries involve constraints on both time and value.

In this chapter, we transform a threshold query on the time series into a two-dimensional bitmap problem so that queries with time and observation constraints can be efficiently executed as two-dimensional spatial range queries. The size of the bitmap is reduced by applying Quadtree decomposition and grouping similar time series into clusters. We demonstrate that due to collectively processing a group of time series and exploiting the pruning power of the Quadtree, the execution of threshold queries is decoupled from the growth of the data (size and number of time series).

6.1 Introduction

Time series are becoming increasingly ubiquitous in many applications across different domains, ranging from finance (e.g., stock information) to science (e.g., sensor readings). Increasingly powerful hardware, e.g., precise instruments or sensors and more powerful computers, lead to ever more and longer time series being recorded. In the simulation sciences, the increasingly powerful supercomputers and the large storage capacity encourage scientists to simulate consistently more

[1]The material of this chapter has been the basis for the SSDBM 2015 paper *RUBIK: Efficient Threshold Queries on Massive Time Series* [150].

detailed models for longer periods of time. Already today scientists are overwhelmed with this deluge of time series data and tools for their efficient analysis are pivotal to enable scientific breakthroughs [106].

Efficiently querying the wealth of time series data is crucial to extract knowledge. In the simulation sciences, analyzing time series enables the scientists to refine their models and to make them more realistic. Unfortunately not all data resulting from a simulation can be analyzed in great detail, but luckily not all data is equally interesting or important either. Typically scientists are interested in particular events and they only need to select a smaller subset of the time series for further analysis. These interesting events can be identified in an exploratory process using threshold queries: searching for time series where the observed value exceeds or falls below a threshold. Subsequently, the interesting subsets can be analyzed in detail with state-of-the-art analysis methods.

The challenge we are addressing in this chapter is to find the interesting time series and, in them, interesting periods of time. More formally, we define a time series as a discrete set of observations $X = x_1, x_2, ...x_n$ at consecutive time steps $t = 1, 2...n$ that are ordered in time. Examples of time series include temperature measurements over time, stock tickers, electrocardiograms and others. Given a set of time series $X_N = \{X_1, X_2, ...X_m\}$ and a query q with a threshold o for the observation as well as an upper and lower bound t_u and t_l with $t_u \geq t_l$ for time, we want to find all time series $X_i \in X_N$ $(1 \leq i \leq m)$ where x_t satisfies the following conditions: $x_t \geq o$ and $t_l \leq t \leq t_u$ (time-bound threshold query). The data management problem is to support efficient time-bound threshold queries, such that query execution scales with an increase in the number of time series and the number of time steps.

Most state-of-the-art time series indexing approaches have been designed to support time series mining tasks, such as testing for similarity and subsequence matching [29, 30, 48, 55, 89, 107]. These indexing techniques typically index time series segments (groups of consecutive time steps) rather than the values at individual time steps, and therefore they cannot be used to answer time-bound threshold queries which need to compare the value at each time step within the time bounds to the threshold value. Bitmap indexes are particularly useful for query-intensive applications and ad-hoc queries on read-only data [34] because they can answer queries by performing efficient bitwise logical operations on the bitmaps. FastBit [167] is a bitmap indexing framework for scientific data that implements Word-Aligned Hybrid Compression (WAH [168]), the state-of-the-art bitmap index compression algorithm which is based on run-length encoding. Compared to other bitmap compression schemes, WAH strikes a good balance between space and time efficiency. However, WAH cannot maintain a one-to-one mapping from a given time step to a specific bit position inside the compressed bit sequence. A condition on the observation value therefore cannot be evaluated in combination with a condition on the time steps without decompressing the bit sequence, which creates a performance overhead for threshold queries.

RUBIK, the approach we develop, compresses the bitmaps that encode the time series without disrupting the mapping from time steps to bit positions inside the compressed representation.

Instead, RUBIK exploits that many time series are, in general and in the simulation sciences in particular, very similar to each other. RUBIK therefore groups similar time series bitmaps together, and compresses them by applying Quadtree decomposition. Queries can then be efficiently executed on the compressed Quadtree as two-dimensional spatial range queries. Exploiting the similarity between time series and representing them using a Quadtree allows for a substantially more effective compression that preserves time resolution and is thus query-efficient. Using the same space budget, RUBIK executes queries up to 9 times faster than the state-of-the-art on a brain simulation micro-benchmark (see Section 6.7.3), and the trend indicates that query execution time with RUBIK will increase considerably slower than related approaches as data sets grow rapidly.

The main contribution of RUBIK consequently lies in transforming a time series threshold query problem into a two-dimensional bitmap problem. Thanks to the representation as a Quadtree, queries with time and observation value predicates can be translated into efficient spatial range queries. Doing so enables the evaluation of both predicates at the same time while the query execution in related work evaluates predicates sequentially, losing early filtering power. Additionally, by grouping time series and decomposing each group collectively using a Quadtree, RUBIK exploits the similarity within and between time series to reduce the size of the bit representation while maintaining high precision.

The remainder of this chapter is organized as follows. In Section 6.2 we review related work on time series indexing and querying methods and in Section 6.3 we motivate RUBIK. We give an overview over RUBIK in Section 6.4 and then discuss in detail the indexing process in Section 6.5 as well as query execution in Section 6.6. Before demonstrating the performance of RUBIK in Section 6.7 we discuss configuration considerations. We finally discuss potential extensions in Section 6.8 and draw conclusions in Section 6.9.

6.2 Related Work

Indexing time series has received considerable attention in recent years [128]. In most analysis tasks the major cost factor is accessing the time series stored on the disk. The general framework consequently is to use a compressed in-memory representation of the time series to answer queries approximately, i.e. a representation providing all the results that satisfy the query (there are no false dismissals) but possibly including some false hits in the answer. The false hits are then dismissed by accessing the candidate time series on the disk. Clearly there is a trade-off between the size of the representation and the number of disk accesses required to refine the approximate answer.

The majority of the work for indexing time series has focused on determining similarity between time series and pattern/subsequence matching. Almost all approaches for similarity-based time series retrieval index the time series with high-dimensional indexes where the number of dimensions corresponds to the number of time steps (length of the time series). To ensure

efficiency – indexes do not scale well to very high dimensions [71] – many approaches to time series mining reduce the number of dimensions, i.e., they divide the time series into segments (along the temporal dimension) and summarize each segment. Summarizing the time steps to reduce dimensionality also compresses the time series.

Data adaptive approaches encode time series individually to minimize the error of the encoding. APCA [89], for example, divides each time series into segments of variable length along the time dimension and encodes each segment with its mean. SAX [30, 97] (and iSAX) builds on the idea of PAA [88] and APCA but uses vertical segmentation of the series (i.e., the definition of breakpoints) to minimize the error for each segment (and thus for the entire time series). SAX also uses a symbolic representation of time series which has the advantage of lower error bound guarantees, i.e., the distance measure on the representation is always equal or smaller than the distance measure on the real data. Many data mining approaches can thus be used directly on the representation as they will only provide false positives, but no false dismissals. To compute the similarity between time series, MVQ [107] identifies frequent subsequences, i.e., codewords, in the time series and assigns each of them a symbol. Each subsequence in time series can then be represented by the codewords (or their symbol) its subsequences resemble the most, yielding a very compact encoding. Because it is difficult to find a suitable resolution (codeword length), MVQ uses several resolutions and organizes the encoding hierarchically, i.e., each frequent subsequence contains shorter frequent subsequences. The distance, or similarity, between two time series is computed based on the weighted frequency of the occurrence of codewords in either time series. GAMPS [55] uses the insight that many time series, particularly resulting from sensors, are similar or at least use similar templates (a subdivision of time series into segments of consecutive time steps). To compress the time series, GAMPS identifies frequent templates among them. It finally represents each time series as a collection of templates (using scaling), considerably reducing their size.

Non data adaptive approaches encode all time series with the same encoding (independent of their individual characteristics). Approaches based on the discrete fourier transformation (DFT) [48] or on discrete cosine transformation (DCT) attempt to preserve the main characteristics of the time series. Fourier transformation is used to extract dominant features (and to reduce the dimensionality) that are then indexed with a spatial index (e.g., an R*-Tree [15]). To perform a subsequence match, the main features from the subsequence are also extracted with DFT and are used to query the spatial index. Chebyshev approximation [29] interpolates time series (or trajectories) with chebyshev polynomials that are easy to compute and have lower bound guarantees. Chebyshev polynomials outperform APCA and PAA for smooth trajectories [29].

Arguably the most renowned indexing approach using compression [168] for scientific data as well as time series is FastBit [167]. FastBit creates bitmap indexes for high-cardinality attributes by applying binning (i.e., discretization) on the time series and encoding the binned values. The resulting bitmaps are compressed using word-aligned hybrid (WAH) compression. Compared to other bitmap compression schemes, WAH strikes a good balance between space and time efficiency. Threshold queries can be executed on the compressed bitmaps but the binning

introduces false positives and so candidate time series have to be retrieved and tested in detail. Detrimental to performance, however, is that the time information has to be indexed separately. Queries on the observation values therefore cannot exploit the pruning power of the time predicate early and vice versa.

6.3 Motivation

Scientific applications produce so much data today that we can no longer afford to analyze all of it in great detail. All the more important are threshold queries to find interesting events in the deluge of time series originating from scientific applications that can then be analyzed in all necessary detail. Executing these threshold queries efficiently is crucial for explorative access to time series data.

6.3.1 Limitations of Related Work

The state-of-the-art for indexing time series has been primarily designed for time series mining, i.e., testing for similarity and subsequence matching. To avoid the curse of dimensionality, time series mining approaches [29, 30, 48, 89, 107] typically reduce the dimensionality by segmenting each time series and approximating each segment with a value a (e.g., in PAA [89] a is the average of all observations in the segment).

Both segmentation and approximation, however, make the aforementioned approaches unsuitable for answering time-bound threshold queries q because they lead to many false positives, and most importantly, can also cause false dismissals. This is because even if the approximation a satisfies the observation threshold of q, this does not guarantee that the individual values at the qualifying time steps satisfy the observation value constraint as well. To avoid false dismissals, the observation threshold o of q needs to be adjusted with the error e, the biggest difference between any observation and the approximation a of its segment. Relaxing the observation threshold o to $o - e$, however, leads to considerably more false positives.

Compression approaches like GAMPS [55] also rely on segmentation and approximation and thus inherit the same problem as the time series mining approaches discussed before. Approaches like FastBit [167], on the other hand, do not index temporal information per se, meaning that a separate index on time steps needs to be built. Clearly the pruning power of the time predicate cannot be used when executing the query using the observation value predicate and vice versa. Further detrimental to the performance of FastBit is that it uses run-length encoding, making the evaluation of the time predicate directly on the compressed data impossible. As a consequence, two separate queries have to be executed on either index directly and the result needs to be combined. Additionally, FastBit and other bitmap-based approaches do not exploit the full potential of compression because they treat time series individually.

As already the work in the context of GAMPS [55] shows, time series frequently have considerable similarity between them and thus compressing them jointly results in improved space efficiency compared to other state-of-the-art approaches. Preserving temporal resolution and avoiding coarse approximation is key to executing time-bound queries without the undue overhead of false positives. As a consequence, RUBIK compresses similar time series jointly while also exploiting similarities within each individual time series, without reducing the temporal resolution (through segmentation).

6.3.2 Motivating Application

Many spatial simulations produce time series as output. In a spatial simulation of material deformation based on a mesh [8], for example, the observed temperature (as well as the change in position) of every mesh vertex is recorded over time. If the temperature increases at a vertex v in the mesh as a result of the deformation, the temperature at neighboring vertices N connected to v in the mesh via edges will increase as well. The time series measuring temperature at each neighboring vertex $n \in N$ of v will consequently resemble each other. Figure 6.1 illustrates this with a temperature snapshot from a deformation simulation: neighboring vertices have a similar temperature (color).

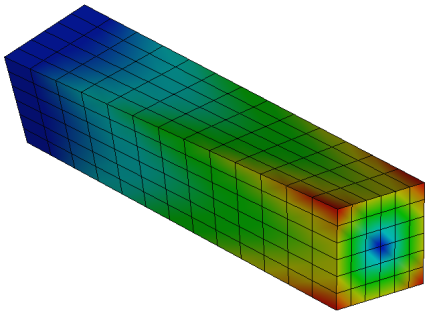


Figure 6.1 – [Best viewed in color] Observing the temperature during a material deformation simulation.

Similarly, the simulation of earthquakes [146] based on meshes produces multitudes of time series with a high degree of similarity. More interestingly to us, and driving the development of RUBIK, are the time series resulting form the simulation of brain activity. We collaborate with neuroscientists in the Blue Brain project (BBP [105]) who simulate the propagation of voltage through very fine-grained models of the neocortex populated with millions of neurons. Also in this scenario, voltage recordings at two neighboring neurons (connected through synapses over which the voltage leaps) are correlated in time and consequently the time series from neighboring neurons will be similar (Figure 6.2).

Crucial for making any approach to time series analysis scale in the future is to compress along both dimensions where time series are growing, i.e., the number of time steps and the number of

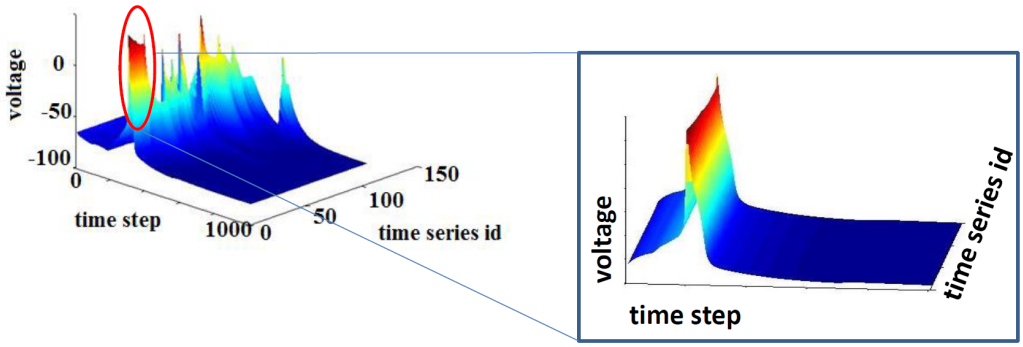


Figure 6.2 – [Best viewed in color] Observing the voltage values during a neuroscience simulation. Time series from neighboring neurons have a high degree of similarity.

time series. When growing the models, scientists not only increase the size (spatial extent) of their models, but also increase their resolution by orders of magnitude [70] as the same shapes will be represented with meshes featuring a substantially higher number of vertices (and edges). The trend to higher resolution models will also lead to more time series (typically one time series per vertex). The resulting time series will, however, have a high degree of similarity because the vertices are closer in space and compression methods must exploit the similarity to achieve good compression in face of growing models.

At the same time, a scalable approach to threshold query execution must compress along the time dimension to address increasingly long-running simulations and the resulting time series. To tackle this growth, exploiting the similarity between consecutive time steps in a time series is pivotal. Time series in a broad range of applications, particularly resulting from the observation or simulation of natural phenomena (e.g., brain simulations, earthquakes, meteorology etc.), are in general smooth, i.e., the observation values of most consecutive time steps only differ by a little, but they may have some massive spikes.

6.4 RUBIK Overview

To enable efficient and scalable threshold query execution in the face of growing time series (number and length), RUBIK takes advantage of the similarity within time series and between time series in general and of time series resulting from the simulation sciences in particular. Given a specific time step t, the values across different time series are similar at t; also inside a given time series, the values between consecutive time steps do not vary much except for spikes (sudden surges, e.g., of voltage, movement etc.). RUBIK exploits the similarity of time series by discretizing them as well as indexing and compressing them with a Quadtree. Threshold queries based on time and observation value predicates used to find interesting time series can then

be translated into efficient two-dimensional range queries on the Quadtree. Crucially, RUBIK preserves the time resolution completely and thereby avoids considerable overhead due to false positives.

More precisely, RUBIK first discretizes/bins the time series along the observation dimension (the time dimension is already implicitly discretized). Each binned time step t_s of a time series is then range encoded: all bins below the observation value o at t_s are set to 1 while the bins greater or equal to o are set to 0. Doing so for all time steps in a time series essentially translates the time series into a two-dimensional bitmap with the area under the curve filled with 1's. The time dimension is implicitly discretized already due to the discrete time steps in simulations or due to epochs in sensor network deployments. Figure 6.3 shows the binning and range-encoding of a time series. By discretizing and range-encoding the time series, RUBIK essentially pre-computes a set of answers for threshold queries that align with the discretization.

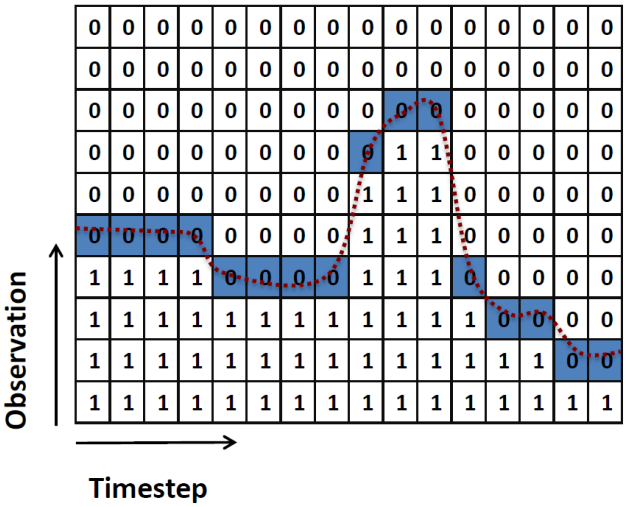


Figure 6.3 – A two-dimensional time series bitmap.

Binning and range-encoding the time series is crucial for compression as otherwise time series almost always differ slightly, rendering efficient compression impossible. Inspired by the Quadtree-based decomposition of bitmap images [33], we notice that Quadtree decomposition can be applied to compress the bitmaps representing the time series. However, this technique compresses each bitmap individually and does not exploit the similarity between the time series. To overcome this limitation, we make clusters of binned time series, i.e., each cluster comprises of several binned time series (in fact the bitmaps corresponding to the time series). Doing so essentially results in the creation of a three-dimensional bitmap where the third dimension is the number of time series in the cluster. Ultimately, we apply the Quadtree decomposition strategy collectively on the whole cluster of bitmaps, that is we hierarchically divide each cluster into four blocks of equal size. The Quadtree decomposition strategy is adapted to accommodate the additional third dimension (number of time series). If a 3D block of bits contains 0's and

1's (in any of the three dimensions), then it is recursively subdivided further in the time and the observation dimension. If, on the other hand, it only contains either 1's or 0's then it is no longer divided. Figure 6.4 illustrates (using the red lines) how the cluster of time series is divided into two along the time and the observation dimension. RUBIK compresses beyond binning and, in case a block only contains 0's and 1's, only stores this information. Using Quadtree decomposition on a 3D bitmap can lead to cases where a block cannot be further subdivided in the time and the observation dimension, yet contains 0's and 1's. To efficiently deal with such mixed blocks and improve space efficiency, RUBIK compresses them with Word Aligned Hybrid compression (WAH [168]).

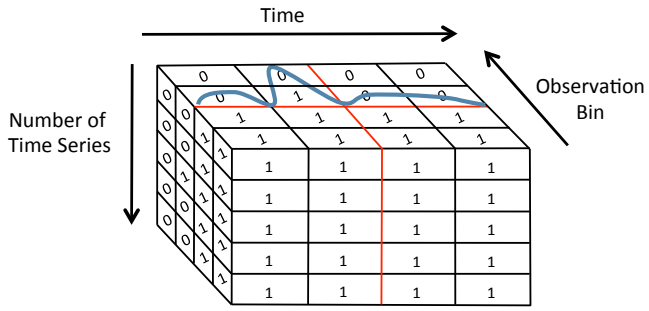


Figure 6.4 – A cluster of time series bitmaps, split along the time and observation dimension with the purpose of identifying blocks enclosing the same bit value.

To execute a threshold query with observation and time predicates, RUBIK translates it into a two-dimensional range query bounded by the time and observation predicates, that can be evaluated on the Quadtree. During query execution, RUBIK first discretizes the observation threshold and then traverses the Quadtree performing query execution on the compressed data. If the bin *b* where the query observation threshold falls is enclosed in blocks that contain only 1's for all the time steps inside the query range, all the time series are returned. If a mixed WAH compressed block is encountered, only the time series that contain a 1 in *b* are returned. Like many related approaches, RUBIK also works on a lossy compression/representation of the time series data, thus retrieves approximate results (containing false positives but no false negatives) and consequently has to verify a subset of the results by reading the exact time series from disk.

6.5 RUBIK Indexing

Indexing with RUBIK has three basic steps: (i) binning & encoding each time series, (ii) grouping similar time series into clusters, and (iii) indexing (as well as compressing) each time series cluster with a Quadtree.

6.5.1 Discretization/Binning

Each time series is discretized along the observation dimension into n bins B where each bin $b \in B$ has an upper and lower boundary bu and bl. RUBIK uses range encoding to encode the discretized values as bitmaps, i.e., given the observation value v at a specific point in time, all bins $b \in B$ with both boundaries smaller than v will be set to 1 whereas all bins with any boundary greater than v will be set to 0. Algorithm 12 illustrates the discretization process with pseudocode. For the purpose of efficiency, RUBIK's implementation wraps the binning/encoding into the building of the Quadtrees, thereby accelerating the indexing process.

Algorithm 12: RUBIK Algorithm for Discretization of one Time Series

Input: ts: array containing a time series
l: number of time steps of time series
$bins_{high}$: array of higher bin boundaries
b: number of bins
Output: bitmap: discretized time series (two-dimensional array)

```
1 for i = 0; i < l; i++ do
2     for j = 0; j < b; j++ do
3         if ts[i] ≥ bins_high[j] then
4             bitmap[i][j] = 1;
5         else
6             bitmap[i][j] = 0;
7 return bitmap
```

The sizes of the bins can vary, depending on the data set, the query workload or both. If, for example, the observation thresholds of queries are frequently in the same range, the precision of the index can be improved if the bins in the range are chosen smaller than outside. Similarly, if the time series have all or most of their observation values in a small range at most points in time, precision can be equally improved by using more (but smaller) bins in this range, thereby taking advantage of RUBIK's ability to use bins of variable width.

6.5.2 Clustering Time Series

To achieve further compression, RUBIK groups similar time series into clusters and indexes each cluster individually. Several approaches have been developed in the past to determine similarity between time series. All of these approaches can be used to group similar time series into clusters. GAMPS [55], for example, clusters time series based on shared subsequences. A simpler approach is to use the inherent local similarity in the time series. As we argued previously, time series from nearby locations in simulation data sets are frequently very similar and so it suffices to use the distance between the locations where the time series have been recorded to compute the clusters.

Figure 6.5 – Coarse-grained discretization for clustering.

RUBIK determines the similarity between time series by using coarse grained binning. In essence RUBIK calculates for each time series a coarse grained binning in both dimensions corresponding to the Quadtree representation at a certain level of resolution (an example is shown in Figure 6.5) and assigns all time series with identical representations to a cluster. A minimum cluster size is set beforehand so that very small clusters (or single time series) are grouped together as one.

The number of clusters (and consequently also their size) is therefore a crucial configuration parameter. With smaller clusters, the time series in them are more similar, and the compression rate for the cluster is higher (despite WAH not compressing small clusters as well as bigger ones). At the same time, however, with more clusters, the data structures (e.g., the Quadtree hierarchy) occupy more space and more Quadtrees need to be queried. Clearly there is an interesting trade-off to be explored between improved compression ratio that the Quadtree decomposition can achieve in each cluster and increased overhead for querying and storing several Quadtrees.

6.5.3 Quadtree Index

RUBIK's indexing process builds a compressed Quadtree-like structure by recursively splitting the cluster of bitmaps representing the time series along the observation and time dimension.

Data Structure

While indexing, RUBIK builds a Quadtree that stores information about the enclosed bits of the bitmap (i.e., whether they are only 1's, only 0's or both) in each node. More precisely, if a node only contains 1's, it is labeled *All 1* and there is no need to store all the enclosed individual bits. Similarly, if a node contains only 0's, it is labeled *All 0* and the enclosed bits do not need to be stored individually. In either case, the node in fact becomes a leaf node as there is no need to partition it further. However, nodes containing both 0's and 1's are labeled *Mixed* and RUBIK splits them in both the observation and time dimension. If a *Mixed* node cannot be split further, RUBIK stores the bit values in a mixed leaf node. To reduce the size of the mixed leaf nodes, RUBIK compresses them with WAH.

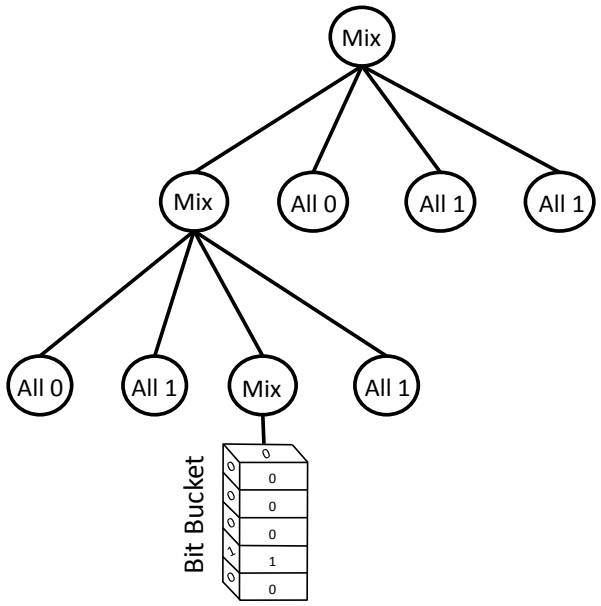


Figure 6.6 – Example Quadtree built by RUBIK in main memory.

Figure 6.6 shows an example of the data structure used in memory: the leaf nodes either store a (compressed) bit bucket, or the information that all bits are 1's or 0's. When writing the Quadtree to disk, RUBIK translates it to a leafless Quadtree where each internal node contains its children. To avoid storing pointers (from nodes to their children), the leafless Quadtree on disk has fixed sized nodes, making nodes directly addressable through the calculation of their address. Mixed nodes are stored in a separate file on disk. A hashmap maps a mixed node in the tree to an offset in the mixed nodes file.

Indexing Algorithm

RUBIK indexes the discretized time series by recursively splitting the cluster of bitmaps (representing time series) along the observation and time dimension into blocks of equal size. If a block only contains either 1's or 0's, it is not split any further and it is stored in a compressed format (labeled as all 1's or all 0's) in the tree structure. The corresponding node in the tree is a leaf node as it will not be split any further. If, on the other hand, the block is mixed, it is split in both dimensions again. The corresponding node in the tree is labeled as mixed. Once a mixed node can no longer be split further because it has length one in either the observation or time dimension, its literal representation is compressed with Word Aligned Hybrid compression (WAH) and is stored in the tree as a leaf node.

Figure 6.7 illustrates how the cluster of discretized, range-encoded time series is split in two steps. Starting with the cluster depicted at the top, the tree is built recursively until the blocks can

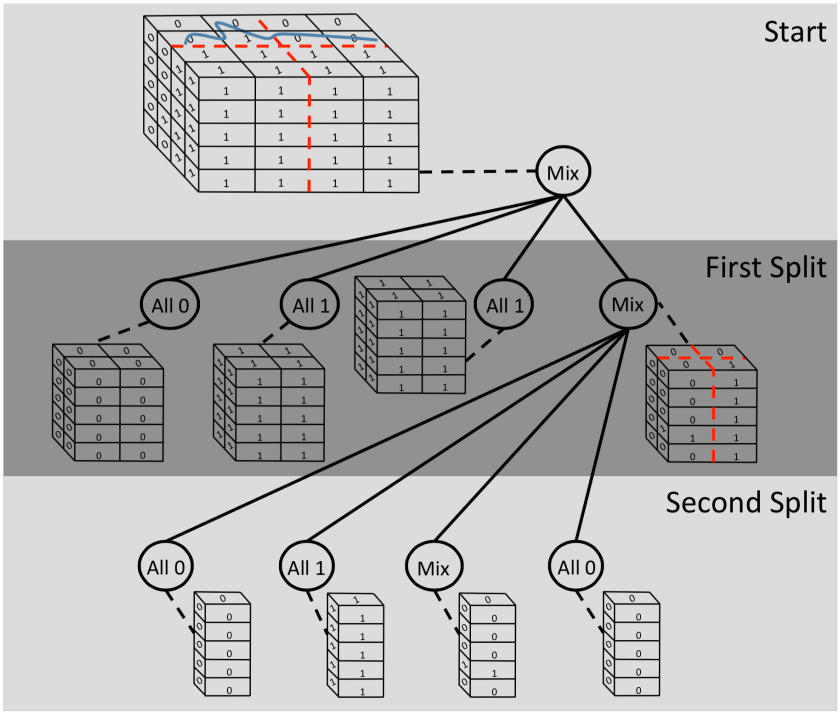


Figure 6.7 – RUBIK splitting the example cluster in two steps.

no longer be divided (after two splits in this example). Each node in the example is connected to the block it represents (dashed line). All blocks are compressed in the Quadtree, except one mixed block that is compressed in the end using WAH.

Algorithm 13 illustrates the process in pseudocode: the cluster of discretized time series is recursively split into smaller blocks until either all blocks are uniform (contain either only 1's or only 0's) or cannot be split any further. The procedure (Algorithm 13) takes as input the cluster and a null root node.

Clearly, determining efficiently whether a block is uniform is crucial. RUBIK computes the type of a block as follows. It maintains two vectors $TimeStepMin[]$ and $TimeStepMax[]$, each having a length equal to the number of time steps. Then, RUBIK scans all the time series in the cluster and, for each time step t_i, stores in $TimeStepMin[t_i]$ the bin number that corresponds to the minimum value for that time step among all the time series, and stores in $TimeStepMax[t_i]$ the bin number that corresponds to the maximum value found for that time step. In other words, the two vectors record the bin numbers of the minimum and maximum value for each time step among all the time series, respectively. Given a block between time steps t_1 and t_2 ($t_1 < t_2$) and observation bins b_1 and b_2 ($b_1 < b_2$), RUBIK iterates over $[t_1, t_2]$. If every $TimeStepMax[t_i]$ is less than or equal to b_1, all the bins of this block have value 0 according the encoding that RUBIK uses; or if every $TimeStepMin[t_i]$ is greater than b_2, all the bins of this block have value 1; otherwise, the block has mixed values of 0 and 1.

Algorithm 13: RUBIK(block, node) Indexing Algorithm

Input: block: three dimensional array containing the clustered time series
parent: parent node in the Quadtree structure
Output: tree: tree structure representing the compressed cluster

```
1  node = new node
2  if (parent == null) then
3      parent = node
4  else
5      attach node to parent
6  if all_one(block) then
7      label node as all 1's
8  else if all_zero(block) then
9      label node as all 0's
10 else
11     if block can be split then
12         label node as mixed
             split block into 4 blocks along time and observation dimension
             RUBIK(block_1, node)
             RUBIK(block_2, node)
             RUBIK(block_3, node)
             RUBIK(block_4, node)
13     else
14         apply WAH compression to block
             label node as mixed
             attach compressed block to node
```

Discussion. In a sense, RUBIK is similar to run-length encoding. Run-length encoding identifies a maximum run (consecutive bits) that have the same bit value and represents them with their bit value and a count instead of storing every bit separately. With RUBIK we attempt to do the same but in three dimensions, i.e., we find 3D areas with the same bit value. To simplify the procedure, we use a Quadtree which essentially predefines the maximum lengths of the runs in two out of the three dimensions. Remaining areas that are mixed (i.e., contain 1's and 0's) are compressed with WAH (in one dimension). This ability of the Quadtree to identify runs in three dimensions by adapting its decomposition strategy for a cluster of bitmaps is the main reason why we chose it as a core component of our approach.

RUBIK could also use an Octree or a KD-Tree to encode the cluster and treat splitting as a three dimensional problem. Either approach would presumably decrease the depth of the tree as splitting in the third dimension (the number of time series) would allow to identify uniform sub-clusters of time series and would eliminate the need to subdivide nodes with bit values differing only in the third dimension. This would therefore also decrease the number of mixed bit buckets that need to be compressed individually. Initial experiments, however, showed that the potential of further compression is small whereas the size of the tree structure increases (additional nodes for splitting along the third dimension). For example, splitting a block that only contains 1's (or 0's) in the third dimension (after it is already split in the observation and time dimension) unnecessarily increases the size of the tree structure without bringing any benefits.

6.6 RUBIK Query Execution

To execute a threshold query, RUBIK maps the query on the observation and time dimension and executes it on the Quadtree which summarizes a cluster of bitmaps (which represent time series). More precisely, RUBIK executes a spatial query which is bounded by the upper and lower bound t_u and t_l of the time predicate in the temporal dimension. In the observation dimension, the upper observation bound of the query is bin b, the bin in which the observation threshold falls. The lower bound is $b - 1$, i.e., the bin just below b.

The first step of query execution consists of performing the spatial query on the Quadtree. The result of this step is a set of nodes that intersect with the query range. The second step of answering a threshold query consists of using the retrieved nodes to determine a set of potential and definite results. Any bitmap (i.e., time series) in the Quadtree that contains a 1 in bin b will definitely be in the final result as a 1 signifies that the actual value is bigger or equal to the upper bound of the bin. Any bitmap in the Quadtree that contains a 0 in bin b but a 1 in bin $b - 1$ potentially is a result as the actual value is bigger than the lower bound and smaller than the upper bound of bin b. Potential results need to be verified by inspecting the actual time series as the precise information is lost due to binning. All results which have a 1 in b will also have a 1 in $b - 1$, i.e., all definite results will also be reported as potential results. To ensure efficient computation of the result, RUBIK thus first identifies the definite result and then the potential result to finally only inspect difference of the two sets in detail (retrieving time series

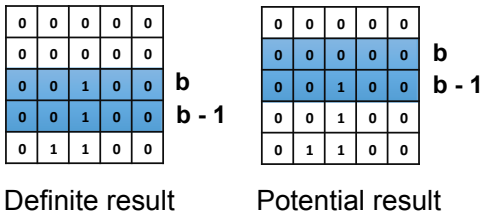


Figure 6.8 – Example of a definite result (left) and a potential result (right).

from disk and analyzing them), i.e., the potential results without the definite ones. Figure 6.8 shows examples of a definite result with a 1 in b (left) and a potential result with a 0 in b and a 1 in $b-1$ (right). While the two examples correspond to the same time series, the threshold queries are different (use different thresholds) resulting in two different spatial queries (shaded).

During query execution, RUBIK benefits significantly from the clustering of the bitmaps, as in the case of uniform nodes all the enclosed bits of the bitmaps have the same value and thus there is no need to examine them one by one. Operating on the WAH compressed bit buckets to determine the values of the individual bits is undoubtly more expensive. However, in case the query is only asking whether any time series exceeds the threshold but does not need to know which one does, decompressing and iterating over a bit bucket is not necessary, as the very fact that a node is labeled mixed implies that there must be at least one 1.

6.7 Experimental Evaluation

In this section we empirically evaluate the performance of RUBIK. We first describe the setup, the methodology and the configuration of the experiments. Then we use a real neuroscience data set to test and evaluate the performance of RUBIK on a real-world example, while we also compare it against FastBit. We focus on FastBit as it is the most broadly used index for the execution of threshold queries on time series. As a final test, we use a synthetically generated data set, where we can control some basic characteristics of the time series.

6.7.1 Experimental Setup

Hardware Configuration. The experiments are run on Red Hat 6.3 machines equipped with 2 quad CPUs AMD Opteron, 64-bit @ 2700 MHz, 32 GB RAM and 4 SAS disks of 300GB (10000 RPM) capacity as storage. We only use one of the disks for the experiments, i.e., no RAID configuration is used.

Software Setup. RUBIK was implemented single-threaded in C++. Additionally a single-threaded application was implemented in C++ on top of the FastBit 2.0.1 API which loads the

data, builds the indexes and queries them. To achieve a fair comparison with RUBIK, FastBit was compiled without memory map support (using defined macros).

Competing Approaches. We experimentally compare FastBit's bitmap indexing approach against RUBIK. We have implemented two different approaches to execute time-bound threshold queries in FastBit. The first option is to use two separate indexes, one for the time dimension and one for the observation value. The second option is to use only one index for the observation value and filter the returned result according to the queried time-bound. This is possible as we are dealing with time-stepped data which allows to map bits to time steps. The additional filtering is applied directly on the bitvectors. The choice between these two options is a trade-off between storage and computation, because the bitmap index for the time dimension used in the first option can be considered as a set of pre-computed filtering masks while in the second option the required mask is computed on-the-fly according to the queried time boundaries.

As discussed in section 6.5, different strategies could be used to choose the bin boundaries. Since the discretization step is common for both FastBit and RUBIK, the same binning strategy is used for both approaches for a fair comparison. All bitmap indexes are range-encoded, so essentially a bitmap index contains bitvectors that are pre-computed answers to threshold queries with a specified precision configured at building time.

All the structures are initialized and the whole index is loaded in memory before querying. Therefore, the measurements do not include I/O operations. Before each experiment, we clear OS caches.

6.7.2 Experimental Methodology

Neuroscience Data Set. The primary data set used in our experiments is obtained from the simulation of brain activity, provided by the neuro-scientists in the Blue Brain Project(BBP [105]). The data set represents the 6th layer of a rat neocortical column and contains 312349 time series. The electrical (action potential) simulation is carried out for 1000 time steps which is thus the length of each time series. In particular, each time series records sequentially the voltage value of a neuron at each time step within a given period of time. An important property of this data set is that time series resulting from neighboring neurons have similar overall patterns. A sample (four time series) of this data set is shown in Figure 6.9. The size of the binary file containing the time series is 1.2GB.

Synthetic Data Set. To test RUBIK further we generated synthetic data that mimics the simulated brain activity. Many models have been proposed for different neuron responses. Some of them are able to reproduce spiking and bursting behavior of known types of cortical neurons based on ordinary nonlinear differential equations. We use a model more similar to our real data, i.e., we use a so-called resonate model. For producing simulated neuron responses we employed the model proposed by A. Watson [162] for temporal sensitivity in visual perception. Specifically,

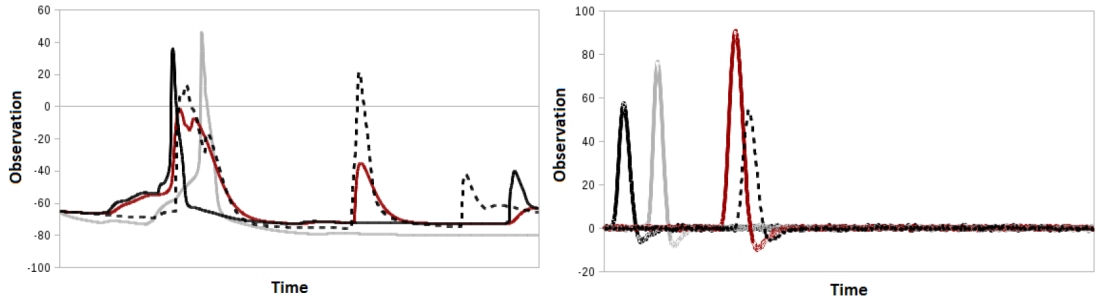


Figure 6.9 – Sample (four time series) of the neuroscience data set (left) and the synthetic data set (right).

we used the impulse response of the model, when the excitation is a spike. A sample (four time series) of this data set is shown in Figure 6.9.

Micro-benchmark. The micro-benchmarks used in our experiments consist of 60 two-dimensional threshold queries which attempt to retrieve the time series of interest. Each threshold query has a time predicate specifying a time period between two time steps and a voltage predicate specifying a value range greater than a value selected randomly from all the possible voltage values.

Approach. To obtain accurate results, the query processing consists of two phases, querying the index and then filtering to eliminate any false positives that the binning has introduced. We are focusing only on the first phase, which essentially calculates two sets of results, one with definite results and one with potential results. The potential results determine which entries need to be verified by testing the full time series. The query execution time reported in all the experiments consists of the time required to count the number of definite results and the number of potential results. The time for identifying the exact location of the candidate results and filtering them is not included.

6.7.3 Comparative Analysis

In this section, we compare the performance of RUBIK and FastBit when a fixed space budget is provided. The space budget allocated to the indexes is fixed to 155MB. We chose this space budget because it allows RUBIK to maintain an accurate in-memory index (128 bins) for our data set. For FastBit, we evaluate three different variants by using two indexes and varying the number of bins dedicated to the time information in $\{10, 25\}$ and by using only one voltage index and removing the returned results that are outside the time range. We refer to them as FastBit<10>, FastBit<25> and FastBitF respectively. First we build the indexes on the brain simulation data set for both RUBIK and FastBit and measure the index sizes to make sure that they respect the space budget. Then we run all the 60 threshold queries in the micro-benchmark and measure the total query execution time for all the approaches.

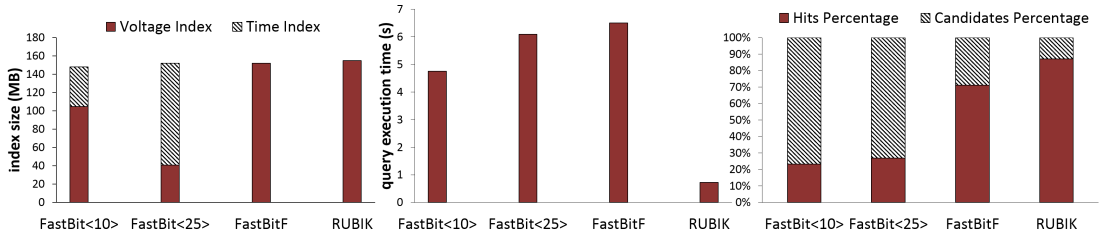


Figure 6.10 – RUBIK and FastBit index sizes (left), execution time (middle) and accuracy (right).

Index Size. Figure 6.10 shows the index size of RUBIK and of the three FastBit variants. All the indexes have a similar size due to the limitation of the space budget. For FastBit the size of the time index increases as the number of bins used for the time information increases, which results in the opposite trend for the voltage index, which is forced to sacrifice its resolution (going from 39 to 14 bins) in order to fit in the allocated budget. FastBitF and RUBIK exploit all the available space budget for the voltage information. As RUBIK compresses more, it uses 128 bins, while FastBitF can only use 54.

Query Execution Time. Figure 6.10 shows the total execution time of 60 threshold queries. The experiment shows that FastBit<25> is slower than FastBit<10>. This is because FastBit<10> does not always use the available time index because of its limited filtering power that renders it useless for some queries. Also, FastBitF is slightly slower than FastBit<25> as building the mask used to filter on-the-fly incurs some processing overhead. Most importantly, however, RUBIK runs 6 to 9 times faster than the different variants of FastBit as it has more pruning power in both the time and the voltage dimension.

Accuracy. Figure 6.10 shows the percentage of hits (results that do not need to be verified) and the percentage of candidates (results that have to be verified by inspecting the time series) with respect to the total number of returned results. We observe that the choice of how the available budget is split among the time and the voltage information in FastBit does not have any significant impact on the accuracy. FastBitF exploits all the available budget for the voltage index, and subsequently filters out some of the candidates. Consequently, FastBitF is the FastBit variant with the best accuracy. However, the accuracy is still not as good as RUBIK's, because FastBitF uses a smaller number of bins for the voltage index.

6.7.4 Scalability Analysis

In this section, we study the impact of parameter configurations and data set characteristics on RUBIK. We vary, respectively, the number of time series, the length of time series, and the number of bins which allows us to test RUBIK with larger data sets, measure how its performance changes accordingly and how it compares to FastBit. For the following experiments, no fixed

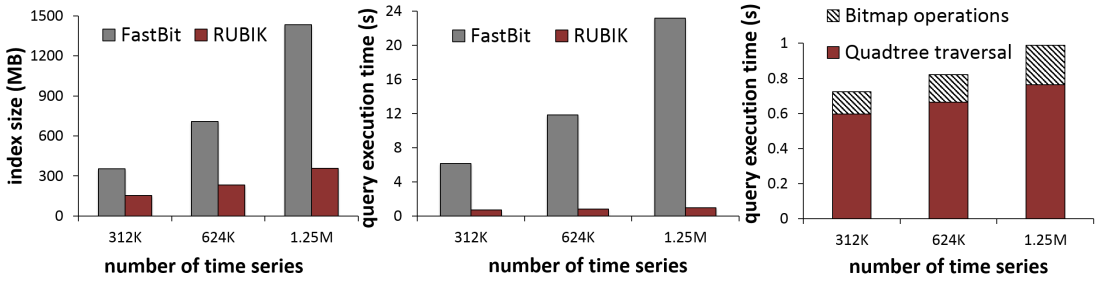


Figure 6.11 – RUBIK and FastBit index sizes (left), execution time (middle) and RUBIK execution time breakdown (right) depending on the number of time series (neuroscience data set).

space budget is used. Instead, we use the same precision for both FastBit and RUBIK, i.e. both approaches use an equal number of bins with the bin boundaries being the same, and we measure the difference in the index size. Our previous experiment shows that the overall best FastBit variant is FastBitF, as for the same space budget it offers increased accuracy, while being only 1.4 times slower than the fastest variant (FastBit<10>). Consequently, in the rest of the experiments we only compare against FastBitF.

Scaling with Data Volume (increase in the number of Time Series) - Neuroscience Data Set

To scale our original data set with respect to the number of time series, we use interpolation. Starting with 312349 time series, we generated two data sets containing double and four times the amount of time series respectively.

Figure 6.11 (left) shows the sizes of the resulting indexes for each data set. RUBIK's index size scales sub-linearly with the number of time series, that is, the compression rate increases as the number of time series increases. Consequently, the larger the size of the original time series data set is, the more compression gain RUBIK achieves.

Figure 6.11 (middle) shows the total query execution time for FastBitF and RUBIK. As RUBIK groups time series, it scales well with the increase in their number because with only one Quadtree traversal the threshold condition is tested on an increasing number of time series. FastBit on the other side has to execute the query on increasingly longer bitvectors. As a result, the achieved speedup over FastBitF increases from 8.5 to 23.

Figure 6.11 (right) shows a breakdown of the total execution time for RUBIK. The *quadtree traversal* stands for the time to perform a 2D spatial range query on the Quadtree (first step of query execution). The *bitmap operations* stands for the time to compute an upper and a lower bound of the number of results (second step of query execution). The quadtree traversal time remains roughly constant, as adding more time series (which fall in the already existing clusters) does not affect the structure of the tree. On the other hand, the bitmap operations time increases

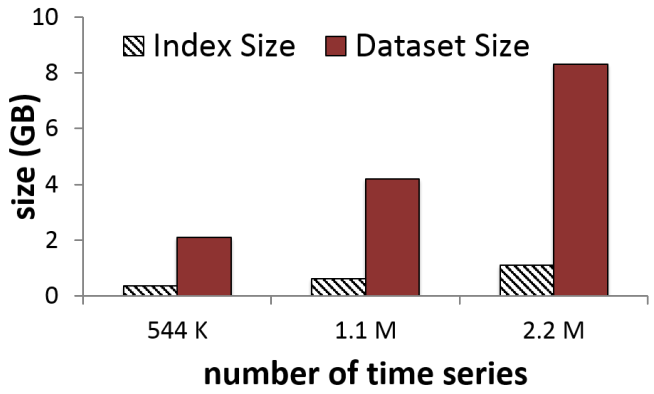


Figure 6.12 – RUBIK index and data sizes depending on the number of time series (synthetic data set).

alongside with the increase in the data size, as each one of the tree nodes is now bigger and consequently the number of returned results that have to be counted is now higher.

Scaling with Data Volume (increase in the number of Time Series) - Synthetic Data Set

To test the performance more thoroughly, we performed the previous experiment for RUBIK using the synthetic data set. We first obtained a base data set of 2.1GB containing 543900 time series, which we scaled up with interpolation. The sizes of the generated data sets are shown in Figure 6.12.

Figure 6.12 shows the size of the indexed data and the index size. Clearly, as the data set size increases, the compression ratio increases as well.

The query execution breakdown of Figure 6.13 exhibits the same trends as in Figure 6.11. As mentioned above, the bitmap operations time increases because the number of results that need to be counted increases as more time series are added.

To perform a comparison with FastBit on a bigger data set, we also built FastBitF on the 8GB data set which contains around 2 million time series. Due to the large size of the raw data and the restricted amount of memory available in the machine used for the experiment, to build the index, FastBit needs to first split the input in smaller partitions. It then builds one index per partition. Figure 6.14 shows the index sizes and the query execution results. For FastBitF the results from different partitions are superimposed. In this case, RUBIK achieves a speedup of 20, while using half the space that FastBitF does (for the same binning).

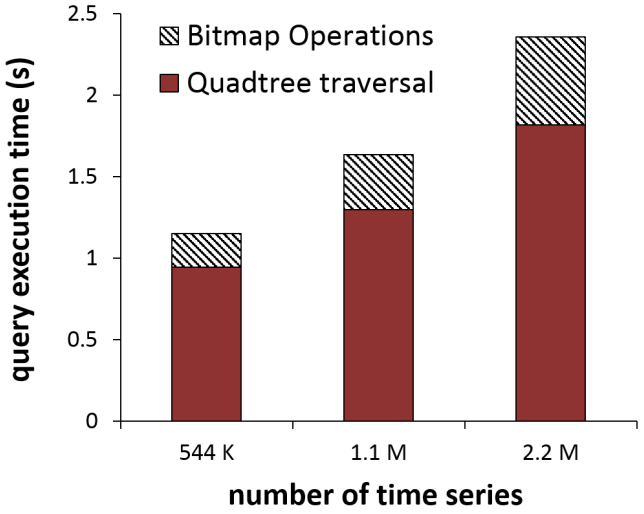


Figure 6.13 – RUBIK execution time breakdown depending on the number of time series (synthetic data set).

Scaling with Temporal Resolution (increase in the number of Time Steps)

To scale our original neuroscience data set with respect to the number of time steps we used interpolation to generate longer time series. Starting with time series that have 1000 time steps, we generated two data sets with time series that have 1999 and 3997 time steps respectively.

Figure 6.15 (left) shows index sizes. We observe that FastBit's WAH compression is able to exploit the increased similarity in the time dimension.

Figure 6.15 (middle) shows the query execution time. We note that the time ranges of the queries in our micro-benchmark are also stretched proportionally to the length of the time series. As FastBitF scales better than RUBIK with the increase in the number of time steps, RUBIK's speedup decreases from 8.5 to 5.5.

Figure 6.15 (right) shows a breakdown analysis of RUBIK's query execution. The increase in the number of time steps results in a deeper Quadtree (as bitmaps become longer in the time dimension and more subdivision steps are required), which causes the quadtree traversal time to increase.

Scaling with Observation Value Resolution (increase in the number of bins)

We build the index of RUBIK on the entire brain simulation data set with three different configurations of number of bins, namely, 128, 256, and 512.

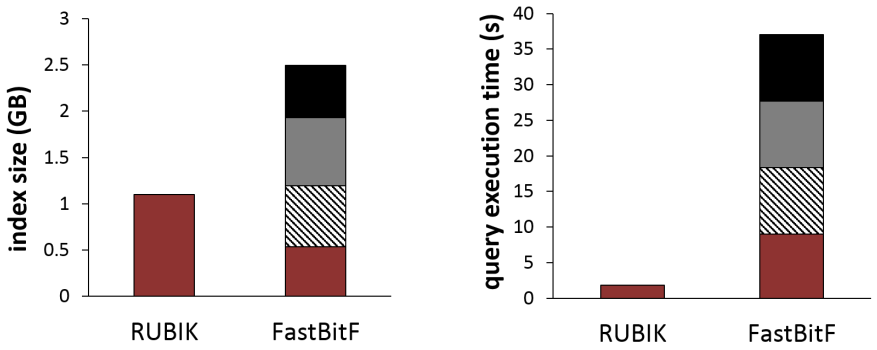


Figure 6.14 – RUBIK and FastBit index sizes (left) and query execution time (right). For FastBitF the results from different partitions are superimposed.

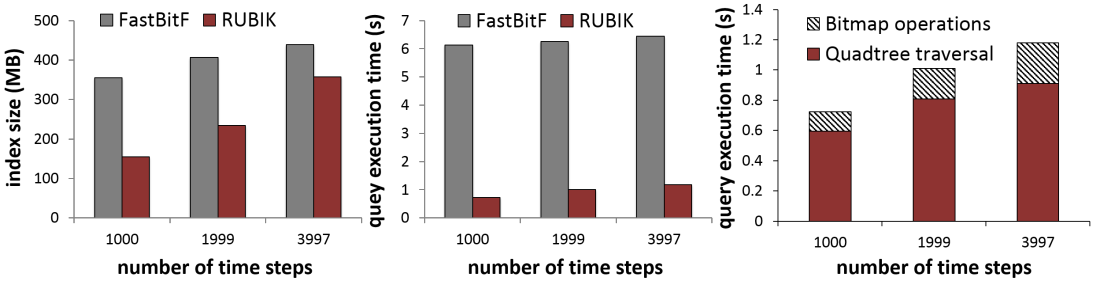


Figure 6.15 – RUBIK and FastBit index sizes (left), execution time (middle) and RUBIK execution time breakdown (right) depending on the number of time steps (neuroscience data set).

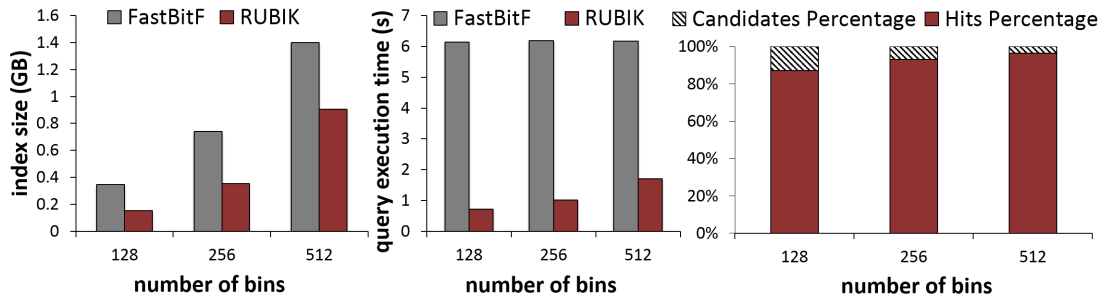


Figure 6.16 – RUBIK and FastBit index sizes (left), execution time (middle) and accuracy (right) depending on the number of bins (neuroscience data set).

Figure 6.16 (left) shows the sizes of the different indexes built for both FastBitF and RUBIK as the number of bins increases. Remarkably, the index size of RUBIK with 256 bins is only slightly bigger than the one that FastBitF builds using only 128 bins. Also, we observe that the index size of FastBitF with 512 bins is 1.4G which is actually bigger than the indexed data itself (1.2G).

In Figure 6.16 (middle) we observe that FastBitF's query execution time is independent of the number of bins. No matter how many bitvectors the bitmap index has, FastBit only retrieves two of them during query execution (the bin b where the threshold falls as well as the immediately lower bin $b-1$). RUBIK on the other side is influenced by the number of bins, because those have an impact on the internal structure of the tree (the bitmaps become longer in the observation dimension).

Figure 6.16 (right) shows the accuracy (i.e., percentage of hits and candidates) of each configuration. Since both approaches use the same binning, their accuracy is the same (but FastBitF requires a larger index). We observe that when more bins are used, the chance of hitting the exact time series increases and conversely the chance of checking the candidates decreases quickly, as higher-resolution binning results in higher indexing precision.

6.7.5 Indexing Time

RUBIK's indexing process consists of the following main steps: after performing one pass over the data, the skeleton of the Quadtree is built in-memory, then WAH compression is applied to the mixed bit buckets and both the compressed bit buckets as well as the Quadtree structure are serialized on disk. Compared to the different FastBit variants, RUBIK requires more time to index the data sets (building time can take from 20 minutes for the smallest data set tested and up to 2 hours for the largest one). However, at the same time, the index needs significantly less space for the same targeted precision. Additionally, in our use cases, indexing is done once the simulation output becomes available and could be performed in an incremental fashion. Alternatively, the different clusters could be indexed in parallel as they are completely independent. Ultimately, we believe there is room for improving RUBIK's indexing performance.

6.8 Discussion

A bitmap index is constructed in three steps: (i) binning/discretization, (ii) encoding, and (iii) compression. The choices for each of these steps are independent from each other. The choice of the encoding scheme is mainly dictated by the targeted queries and the data. RUBIK uses range encoding to encode the discretized values. However, the quadtree-based compression scheme proposed in RUBIK is not tied to a particular encoding scheme. The time series values could be encoded using other bitmap encoding schemes proposed in the literature (e.g., interval encoding [32]). The encoding scheme simply determines how the bitmap is populated with 0's and 1's, or in other words which bits are set to 0 and which bits are set to 1 in each bitmap of the index. The Quadtree compression strategy then takes these 0's and 1's and combines them in blocks.

Quadtree compression exploits uniform blocks in the bitmap index, thus its performance depends directly on the presence of such uniform blocks. As a result, the effectiveness varies for different

groupings of the data values as well as for different orderings of the data values within each group, since both the grouping and the ordering affect uniform blocks. In this work, we propose a simple clustering algorithm that groups the time series in a way that enables effective compression. Investigating how to re-order and partition data tuples in order to produce bitmaps that are compressible with Quadtrees, is an interesting avenue that could potentially enable using Quadtree compression for data other than time series.

In this work, we show the benefits of RUBIK for what we call time-bound threshold queries of the form: $x_t \geq o$ and $t_l \leq t \leq t_u$. The implementation of RUBIK can be extended to support more operators in the value domain other than $\geq$, that is $<, \leq$, and $>$. It can also be extended to support two-sided range conditions and equality conditions on the time series values. Even though RUBIK can support queries that have a condition only in the time or only in the value dimension (and leave the other dimension unbounded), to fully take advantage of the pruning power that the Quadtree structure offers in both dimensions, it is preferable to use RUBIK for evaluating queries with 2D range conditions.

Finally, we note that RUBIK's compression scheme can be seen as complementary to run-length encoding (WAH). Essentially, RUBIK first attempts to find correlations in more dimensions within the bitmap. When that fails, RUBIK resorts to WAH to compress bits in one dimension.

6.9 Chapter Summary

In this chapter, we present RUBIK, a novel approach for indexing time series data. RUBIK transforms threshold queries on time series into a two-dimensional bitmap problem. By decomposing the time series using a Quadtree, RUBIK reduces the size of the bit representation while maintaining high precision as otherwise the number of false positives imposes an undue penalty on the query execution. Thanks to the representation as a Quadtree, queries with time and observation value predicates can be translated into efficient spatial range queries.

The Quadtree representation along with the use of WAH for compression also exploit that time series in many application domains and particularly in the simulation sciences are often similar to each other. By using both, Quadtree and WAH, RUBIK can efficiently compress similar time series and scales particularly well with time series resulting from increasingly detailed simulation models as our experiments show.

Crucially, our experimental evaluation shows that, because RUBIK can collectively represent and process a group of time series as well as exploit the pruning power of the Quadtree, it outperforms the state-of-the-art by a factor of 6 to 23 for query execution while producing a more space-efficient index.

7 Conclusion and Outlook

Spatial and temporal data exploration leads to knowledge discovery. However, there is a gap between the capabilities of existing data management approaches and the requirements of spatial and temporal data exploration. To bridge this gap, in this thesis we advocate for spatial query operators that leverage GPU rendering and a workload-aware design of access methods.

To enable interactive exploration, we decompose spatial query operators into graphics primitives executed on graphics hardware (GPU) (Chapter 3). To accelerate spatial data processing, we improve the approximation precision of existing bounding predicates by subtracting out bounded areas that are empty (Chapter 4). To enable ad-hoc exploration of multiple spatial data sets, we design an index structure that leverages both spatial proximity as well as data set membership (Chapter 5 - Part I), and propose to incrementally index subsets from different data sets that are frequently queried together (Chapter 5 - Part II). To enable scalable time series exploration, we introduce an access method that allows evaluating queries with both value and time constraints, and leverages similarity within and across time series (Chapter 6).

We conclude with a brief discussion on research directions related to this thesis.

7.1 Looking Ahead

Computer Graphics and Spatial Databases. Chapter 3 showcases the utility of computer graphics techniques in the context of spatial data processing. Since spatial databases rely on the same primitive types (geometric objects such as points and polygons) and operations that are similar to the ones used in graphics (e.g., spatial selections and containment tests), we expect further opportunities to exploit advanced graphics techniques and hardware in the design of new spatial data management solutions. For instance, spatial joins in three dimensions can be framed as a collision detection problem in graphics. Detecting collisions between graphical objects in real-time is critical for several graphics applications such as virtual reality, and it has thus attracted a lot of attention in the computer graphics community, resulting in techniques that could be exploited to evaluate spatial joins in three dimensions at interactive speeds.

Approximation-based Spatial Data Processing. Spatial approximations have been traditionally used in spatial databases to accelerate processing of complex geometric operations. However, approximations are only used as a first filtering step to determine a set of candidate spatial objects that may fulfill the query condition. To provide accurate results, the exact geometries of the candidate objects are tested against the query condition. Nevertheless, in many applications (such as visualization tools) approximate results are often sufficient. Besides, real-world geospatial data is inherently imprecise. GPS positions are typically accurate to within a 4.9 m radius under open sky [156]. Geographical region boundaries are usually fuzzy, in the sense that adjacent regions are separated by an extended zone (e.g., a street surface) rather than a one-dimensional line. Given the uncertainty associated with spatial data and the relaxed precision requirements of many applications, we need spatial data processing techniques that omit exact geometric tests and provide final answers solely on the basis of (fine-grained) approximations. The employed approximations should allow to control the precision and trade accuracy for performance.

Exploiting Large Main Memory Capacity of Modern Hardware. Traditional spatial query processing approaches rely on a two-step "filter and refine" strategy [54] where the query is first evaluated using spatial approximations and then false matches are eliminated with exact geometric tests. This strategy is based on the underlying assumption that main memory is scarce, and thus sacrifices precision for a more compact approximation. However, modern machines have large main memory sizes that can go up to multiple terabytes. Therefore, we can afford to increase the precision (and size) of spatial approximations in exchange for better performance. We believe that developing new fine-grained spatial approximations alongside with query-efficient index structures that maintain these approximations in main memory is a promising research direction.

Distance-based Error Bounds for Approximation-based Spatial Data Processing. In chapter 3, we employ a distance-based error bound that controls the maximum distance between the partners of a false positive/negative join pair. To the best of our knowledge, we are the first to introduce a formal distance-based error bound for spatial data processing. Spatial data processing techniques evaluate relations between objects in space (e.g., intersection, containment). Therefore, we believe that for meaningful analyses, approximate techniques should provide bounds on the spatial distance between false or missing and exact results.

GPU Rasterization for Real-Time Approximation. As we discuss in Chapters 3 and 5 (Part II), in data exploration applications users can change dynamically not only the parameters of their queries, but also the input data. Therefore, it is not always feasible to rely on pre-processing for efficiency. We need methods to process spatial data fast on-the-fly. The use of GPU rasterization is a promising step in that direction. Being a crucial component of the graphics rendering pipeline, rasterization is natively supported by GPUs and is performed at interactive speeds. By rasterizing geometric primitives (i.e., polygons) we convert them into a collection of pixels on-the-fly. This collection of pixels essentially forms a fine-grained uniform grid approximation of the geometric primitive and can be leveraged to process queries efficiently (as we show in Chapter 3).

In Chapter 4, we use Pareto optimality to determine the area that is empty around the corner of a Minimum Bounding Box. For two-dimensional data, instead of Pareto optimality, GPU rasterization could be used to determine empty areas within a Minimum Bounding Box. This would significantly reduce the time for generating and updating Clipped Bounding Boxes.

Clipped Bounding Boxes as first-class citizens. In Chapter 4, we advocate for clipping Minimum Bounding Boxes (MBBs) and plug the clipped boxes into different R-tree variants. Currently, we perform the clipping as an afterthought during R-tree construction, leaving the construction algorithms unmodified. The main procedures performed during R-tree construction are (i) choosing the most suitable sub-tree at every level to accommodate a newly inserted object, and (ii) splitting objects among two nodes upon a node overflow [15]. The goal of those procedures is to produce nodes with both minimal coverage and overlap, which are measured on the basis of MBBs. However, there is the opportunity to produce a better index structure if we modify the R-tree construction algorithms to take into account the possibility of clipping empty MBB corners. Alternatively, we could design an entirely new index structure on the basis of Clipped Bounding Boxes.

Bibliography

[1] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 29–42, 2013.

[2] Alok Aggarwal, J. S. Chang, and K. Yap Chee. Minimum area circumscribing Polygons. *The Visual Computer*, 1(2):112–117, 1985.

[3] Danial Aghajarian, Satish Puri, and Sushil Prasad. GCMF: An Efficient End-to-end Spatial Join System over Large Polygonal Datasets on GPGPU Platform. In *Proceedings of the SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2016.

[4] Ablimit Aji, Fusheng Wang, Hoang Vo, Rubao Lee, Qiaoling Liu, Xiaodong Zhang, and Joel H. Saltz. Hadoop-GIS: A High Performance Spatial Data Warehousing System over MapReduce. *PVLDB*, 6(11):1009–1020, 2013.

[5] Volkan Akcelik, Jacobo Bielak, et al. High resolution forward and inverse earthquake modeling on terascale computers. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, page 52, 2003.

[6] Varol Akman, William Randolph Franklin, Mohan Kankanhalli, and Chandrasekhar Narayanaswami. Geometric computing and uniform grid technique. *Computer-Aided Design*, 21(7):410–420, 1989.

[7] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[8] Guillaume Anciaux, Srinivasa B. Ramisetti, and Jean François Molinari. A finite temperature bridging domain method for MD-FE coupling and application to a contact problem. *Computer Methods in Applied Mechanics and Engineering*, 205-208(1):204–212, 2012.

[9] Gennady Andrienko, Natalia Andrienko, Christophe Hurter, Salvatore Rinzivillo, and Stefan Wrobel. Scalable Analysis of Movement Data for Extracting and Exploring

Significant Places. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 19(7):1078–1094, 2013.

[10] Gennady Antoshenkov. Byte-aligned bitmap compression. In *Proceedings of the Conference on Data Compression (DCC)*, 1995.

[11] Naiyong Ao, Fan Zhang, Di Wu, Douglas S. Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Sheng Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[12] Lars Arge, Mark de Berg, Herman J. Haverkort, and Ke Yi. The Priority R-tree: A Practically Efficient and Worst-case Optimal R-tree. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 347–358, 2004.

[13] Ira Assent, Ralph Krieger, Farzad Afschari, and Thomas Seidl. The TS-tree: Efficient Time Series Search and Retrieval. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 252–263, 2008.

[14] Leilani Battle, Michael Stonebraker, and Remco Chang. Dynamic reduction of query result sets for interactive visualization. In *Proceedings of the IEEE International Conference on Big Data*, pages 1–8, 2013.

[15] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[16] Norbert Beckmann and Bernhard Seeger. *A Benchmark for Multidimensional Index Structures*. http://www.mathematik.uni-marburg.de/~seeger/rrstar/.

[17] Norbert Beckmann and Bernhard Seeger. A Revised R*-tree in Comparison with Related Index Structures. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 799–812, 2009.

[18] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, 1975.

[19] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 1996.

[20] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[21] Laurynas Biveinis, Simonas Šaltenis, and Christian S. Jensen. Main-memory operation buffering for efficient R-tree update. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 591–602, 2007.

[22] Stephan Börzsönyi, Donald Kossman, and Konrad Stocker. The skyline operator. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 421–430, 2001.

[23] Sotiris Brakatsoulas, Dieter Pfoser, and Yannis Theodoridis. Revisiting R-tree construction principles. In *Proceedings of the European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 149–162, 2002.

[24] Sebastian Breß, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. GPU-Accelerated Database Systems: Survey and Open Challenges. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[25] Karl Bringmann, Tobias Friedrich, and Patrick Klitzke. Two-dimensional Subset Selection for Hypervolume and Epsilon-Indicator. In *Proceedings of the Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pages 589–596, 2014.

[26] Thomas Brinkhoff, H.-P. Kriegel, and Ralf Schneider. Comparison of approximations of complex objects used for approximation-based query processing in spatial database systems. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 40–49, 1993.

[27] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 197–208, 1994.

[28] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins Using R-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 237–246, 1993.

[29] Yuhan Cai and Raymond Ng. Indexing Spatio-Temporal Trajectories with Chebyshev Polynomials. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2004.

[30] Alessandro Camerra, Themis Palpanas, Jin Shieh, and Eamonn Keogh. iSAX 2.0: Indexing and Mining One Billion Time Series. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, 2010.

[31] Alessandro Camerra, Jin Shieh, Themis Palpanas, Thanawin Rakthanmanon, and Eamonn Keogh. Beyond one billion time series: Indexing and mining very large time series collections with iSAX2+. *Knowledge and Information Systems*, 39, 2014.

[32] Chee-Yong Chan and Yannis E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–226, 1999.

[33] Yung-Kuan Chan. Block image retrieval based on a compressed linear quadtree. *Image and Vision Computing*, 22(5):391–397, 2004.

[34] Surajit Chaudhuri and Umeshwar Dayal. An Overview of Data Warehousing and OLAP Technology. *ACM SIGMOD Record*, 26:65–74, 1997.

[35] Chicago Open Data. https://data.cityofchicago.org/.

[36] Fernando Chirigati, Harish Doraiswamy, Theodoros Damoulas, and Juliana Freire. Data Polygamy: The Many-Many Relationships Among Urban Spatio-Temporal Data Sets. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1011–1025, 2016.

[37] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. Automatic Contention Detection and Amelioration for Data-intensive Operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[38] Using clipper and poly2tri together for robust triangulation. https://github.com/raptor/clip2tri, 2015.

[39] Stanley Coren, Lawrence M. Ward, and James T. Enns. *Sensation and Perception*. Wiley, 2003.

[40] Oracle Corporation. Oracle Spatial and Graph: Advanced data management. Technical report, Oracle, 2014.

[41] Ondrej Danko and Tomáš Skopal. Elliptic indexing of multidimensional databases. In *Proceedings of the Australasian Database Conference (ADC)*, pages 85–94, 2009.

[42] Judith R. Davis. IBM's DB2 Spatial Extender: Managing Geo-Spatial Information with the DBMS. *IBM White Paper*, 1998.

[43] Harish Doraiswamy, Nivan Ferreira, Theodoros Damoulas, Juliana Freire, and Cláudio T. Silva. Using Topological Analysis to Support Event-Guided Exploration in Urban Data. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 20(12):2634–2643, 2014.

[44] Harish Doraiswamy, Juliana Freire, Marcos Lage, Fabio Miranda, and Cláudio T. Silva. Spatio-Temporal Urban Data Analysis: A Visual Analytics Perspective. *IEEE Computer Graphics and Applications (CG&A)*, 38(5):26–35, 2018.

[45] Harish Doraiswamy, Eleni Tzirita Zacharatou, Fábio Miranda, Marcos Lage, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. Interactive Visual Exploration of Spatio-Temporal Urban Data Sets using Urbane. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1693–1696, 2018.

[46] Harish Doraiswamy, Huy T. Vo, Cláudio T. Silva, and Juliana Freire. A GPU-based index to support interactive spatio-temporal queries over historical data. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 1086–1097, 2016.

[47] Ahmed Eldawy. SpatialHadoop: Towards flexible and scalable spatial processing using MapReduce. In *SIGMOD, PhD Symposium*, pages 46–50, 2014.

[48] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast Subsequence Matching in Time-series Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[49] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *PVLDB*, 3(1-2):670–680, 2010.

[50] Yi Fang, Marc Friedman, Giri Nair, Michael Rys, and Ana-Elisa Schmid. Spatial indexing in microsoft SQL server 2008. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1207–1216, 2008.

[51] Nivan Ferreira, Marcos Lage, Harish Doraiswamy, Huy Vo, Luc Wilson, Heidi Werner, Muchan Park, and Cláudio Silva. Urbane: A 3D framework to support data driven decision making in urban development. In *Proceedings of the IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 97–104, 2015.

[52] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1990.

[53] Foursquare. https://foursquare.com/about.

[54] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[55] Sorabh Gandhi, Suman Nath, Subhash Suri, and Jie Liu. GAMPS: Compressing Multi Sensor Data by Grouping and Amplitude Scaling. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2009.

[56] Yván J. García R, Mario A. Lopez, and Scott T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Proceedings of the ACM International Symposium on Advances in Geographic Information Systems (GIS)*, pages 163–164, 1998.

[57] Making the most detailed tweet map ever. https://blog.mapbox.com/making-the-most-detailed-tweet-map-ever-b54da237c5ac.

[58] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 215–226, 2004.

[59] Goetz Graefe and Harumi Kuno. Adaptive Indexing for Relational Keys. In *Workshops Proceedings of the IEEE International Conference on Data Engineering (ICDEW)*, 2010.

[60] Goetz Graefe and Harumi Kuno. Self-selecting, Self-tuning, Incrementally Optimized Indexes. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 371–381, 2010.

[61] Ronald L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972.

[62] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[63] Jim Gray, David T. Liu, Maria Nieto-Santisteban, Alex Szalay, David J. DeWitt, and Gerd Heber. Scientific Data Management in the Coming Decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[64] Jim Gray, Prakash Sundaresan, Susanne Englert, Ken Baclawski, and Peter J. Weinberger. Quickly Generating Billion-record Synthetic Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[65] Oliver Gunther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 598–605, 1989.

[66] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record*, 14(2):47–57, 1984.

[67] Ralf Hartmut Güting. An introduction to spatial database systems. *The VLDB Journal*, 3(4):357–399, 1994.

[68] Mark Harrower and Cynthia A. Brewer. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[69] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 511–524, 2008.

[70] Thomas Heinis, Farhan Tauheed, and Anastasia Ailamaki. Spatial Data Management Challenges in the Simulation Sciences. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, 2014.

[71] Joseph M. Hellerstein, Elias Koutsoupias, and Christos H. Papadimitriou. On the Analysis of Indexing Schemes. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, 1997.

[72] Tony Hey, Stewart Tansley, and Kristin Tolle, editors. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, Redmond, Washington, 2009.

[73] The Human Brain Project. https://www.humanbrainproject.eu.

[74] IBM Informix Dynamic Server v12.1 Information Center. *IBM Informix R-tree Index, User's Guide*, 2013.

[75] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database Cracking. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2007.

[76] Stratos Idreos, Stefan Manegold, Harumi A. Kuno, and Goetz Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9):586–597, 2011.

[77] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. Overview of Data Exploration Techniques. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 277–281, 2015.

[78] Jean-François Im, Félix Giguère Villegas, and Michael J. McGuffin. VisReduce: Fast and responsive incremental information visualization of large datasets. In *Proceedings of the IEEE International Conference on Big Data*, pages 25–32, 2013.

[79] Alfred Inselberg and Bernard Dimsdale. Parallel coordinates. In *Human-Machine Interactive Systems*, pages 199–233. Springer, 1991.

[80] Chris L. Jackins and Steven L. Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3), 1980.

[81] Edwin H. Jacox and Hanan Samet. Spatial Join Techniques. *ACM Transactions on Database Systems (TODS)*, 32(1), 2007.

[82] H. Vi Jagadish. Spatial search with polyhedra. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 311–319, 1990.

[83] Uwe Jugel, Zbigniew Jerzak, Gregor Hackenbroich, and Volker Markl. M4: A Visualization-Oriented Time Series Data Aggregation. *PVLDB*, 7(10):797–808, 2014.

[84] Niranjan Kamat, Prasanth Jayachandran, Karthik Tunga, and Arnab Nandi. Distributed and interactive cube exploration. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 472–483, 2014.

[85] Ibrahim Kamel and Christos Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 500–509, 1994.

[86] Manos Karpathiotakis, Miguel Branco, Ioannis Alagiannis, and Anastasia Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7(12):1119–1130, 2014.

[87] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 369–380, 1997.

[88] Eamonn Keogh, Kaushik Chakrabarti, Michael Pazzani, and Sharad Mehrotra. Dimensionality Reduction for Fast Similarity Search in Large Time Series Databases. *Knowledge and Information Systems*, 3(3), 2001.

[89] Eamonn J. Keogh, Kaushik Chakrabarti, Sharad Mehrotra, and Michael J. Pazzani. Locally Adaptive Dimensionality Reduction for Indexing Large Time Series Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2001.

[90] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony D. Nguyen, Tim Kaldewey, Victor W. Lee, Scott A. Brandt, and Pradeep Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 339–350, 2010.

[91] Ravi Kanth V. Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in oracle spatial: a comparison using GIS data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 546–557, 2002.

[92] YongChul Kwon, Dylan Nunley, Jeffrey P. Gardner, Magdalena Balazinska, Bill Howe, and Sarah Loebman. Scalable clustering algorithm for N-body simulations in a shared-nothing cluster. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 132–150, 2010.

[93] Mong Li Lee, Wynne Hsu, Christian S. Jensen, Bin Cui, and Keng Lik Teo. Supporting frequent updates in R-trees: a bottom-up approach. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 608–619, 2003.

[94] Daniel Lemire, Gregory Ssi Yan Kai, and Owen Kaser. Consistently faster and smaller compressed bitmaps with Roaring. *Software Practice and Experience*, 46(11):1547–1569, 2016.

[95] Scott T. Leutenegger, Mario Lopez, and Jeffrey Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 497–506, 1997.

[96] Zhisheng Li, Ken C. K. Lee, Baihua Zheng, Wang-Chien Lee, Dik Lee, and Xufa Wang. IR-Tree: An Efficient Index for Geographic Document Search. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 23(4):585–599, 2011.

[97] Jessica Lin, Eamonn Keogh, Li Wei, and Stefano Lonardi. Experiencing SAX: A Novel Symbolic Representation of Time Series. *Data Mining and Knowledge Discovery*, 15(2), 2007.

[98] Lauro Lins, James T. Klosowski, and Carlos Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 19(12):2456–2465, 2013.

[99] Zhicheng Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 20(12):2122–2131, 2014.

[100] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum (CGF)*, 32:421–430, 2013.

[101] David B. Lomet and Betty Salzberg. The hB-tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM Transactions on Database Systems (TODS)*, 15(4):625–658, 1990.

[102] Nikos Mamoulis. *Spatial Data Management*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[103] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Yannis Theodoridis. *R-Trees: Theory and Applications*. Springer, 2005.

[104] Mapd technology. https://www.mapd.com/.

[105] Henry Markram. The Blue Brain Project. *Nature Reviews Neuroscience*, 7(2):153–160, 2006.

[106] Henry Markram et al. Introducing the Human Brain Project. *Procedia Computer Science*, 7:39–42, 2011.

[107] Vasileios Megalooikonomou, Qiang Wang, Guo Li, and Christos Faloutsos. A Multiresolution Symbolic Representation of Time Series. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2005.

[108] Fabio Miranda, Harish Doraiswamy, Marcos Lage, Kai Zhao, Bruno Gonçalves, Luc Wilson, Mondrian Hsieh, and Cláudio T. Silva. Urban Pulse: Capturing the Rhythm of Cities. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 23(1):791–800, 2017.

[109] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications (CG&A)*, 14(4):23–32, July 1994.

[110] *MySQL 5.0 Reference Manual (11.5 Extensions for Spatial Data)*, 2015.

[111] Tahora H. Nazer, Guoliang Xue, Yusheng Ji, and Huan Liu. Intelligent Disaster Response via Social Media Analysis - A Survey. *ACM SIGKDD Explorations Newsletter*, 19(1):46–59, 2017.

[112] Nvidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[113] NYC Open Data. http://data.ny.gov.

[114] Marc Olano and Trey Greer. Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS, pages 89–95, 1997.

[115] Yahoo labs. https://webscope.sandbox.yahoo.com/.

[116] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 1994.

[117] Thomas Ortner, Johannes Sorger, Harald Steinlechner, Gerd Hesina, Harald Piringer, and Eduard Gröller. Vis-A-Ware: Integrating Spatial and Non-Spatial Visualization for Visibility-Aware Urban Planning. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 23(2):1139–1151, Feb 2017.

[118] Cícero A. L. Pahins, Sean A. Stephens, Carlos Scheidegger, and João L. D. Comba. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 23(1):671–680, 2017.

[119] Themis Palpanas. Big Sequence Management: A glimpse of the Past, the Present, and the Future. In *Proceedings of the International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*, volume 9587, pages 63–80. Springer, 2016.

[120] Varun Pandey, Andreas Kipf, Dimitri Vorona, Tobias Mühlbauer, Thomas Neumann, and Alfons Kemper. High-Performance Geospatial Analytics in HyPerSpace. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 2145–2148, 2016.

[121] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases (SSTD)*, 2001.

[122] Jignesh M. Patel and David J. DeWitt. Partition Based Spatial-Merge Join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1996.

[123] Mirjana Pavlovic, Thomas Heinis, Farhan Tauheed, Panagiotis Karras, and Anastasia Ailamaki. TRANSFORMERS: Robust spatial joins on non-uniform data distributions. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 673–684, 2016.

[124] Mirjana Pavlovic, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. QUASII: QUery-Aware Spatial Incremental Index. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 325–336, 2018.

[125] Mirjana Pavlovic, Eleni Tzirita Zacharatou, Darius Sidlauskas, Thomas Heinis, and Anastasia Ailamaki. Space Odyssey: Efficient Exploration of Scientific Data. In *Proceedings of the ACM SIGMOD/PODS International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*, 2016.

[126] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. *SIGGRAPH Proceedings of the Annual conference on Computer Graphics and Interactive Techniques*, 22(4):17–20, 1988.

[127] PostGIS: Spatial and geographic objects for PostgreSQL. http://postgis.net/.

[128] John F. Roddick, Kathleen Hornsby, and Myra Spiliopoulou. An Updated Bibliography of Temporal, Spatial, and Spatio-temporal Data Mining Research. In *Proceedings of the International Workshop on Temporal, Spatial, and Spatio-Temporal Data Mining (TSDM)*. Springer, 2001.

[129] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[130] Roeland Scheepens, Niels Willems, Huub van de Wetering, Gennady Andrienko, Natalia Andrienko, and Jarke J. van Wijk. Composite Density Maps for Multivariate Trajectories. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 17(12):2518–2527, 2011.

[131] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[132] Jonathan Richard Shewchuk. Delaunay Refinement Algorithms for Triangular Mesh Generation. *Computational Geometry: Theory and Applications*, 22(1-3):21–74, 2002.

[133] Ben Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[134] Dave Shreiner, Graham Sellers, John M. Kessenich, and Bill M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[135] Darius Šidlauskas, Sean Chester, Eleni Tzirita Zacharatou, and Anastasia Ailamaki. Improving Spatial Data Processing by Clipping Minimum Bounding Boxes. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 425–436, 2018.

[136] Darius Šidlauskas, Christian S. Jensen, and Simonas Šaltenis. A Comparison of the Use of Virtual Versus Physical Snapshots for Supporting Update-intensive Workloads. In *Proceedings of the ACM SIGMOD/PODS International Workshop on Data Management on New Hardware (DaMoN)*, 2012.

[137] Bogdan Simion, Daniel N. Ilha, Angela Demke Brown, and Ryan Johnson. The price of generality in spatial indexing. In *Proceedings of the ACM SIGSPATIAL International Workshop on analytics for Big Geospatial Data (BigSpatial)*, 2013.

[138] Emmanuel Stefanakis, Yannis Theodoridis, Timos Sellis, and Yuk-Cheung Lee. Point Representation of Spatial Objects and Query Window Extension: A new Technique for Spatial Access Methods. *International Journal of Geographical Information Science (IJGIS)*, 11(6), 1997.

[139] Chris Stolte and Pat Hanrahan. Polaris: A System for Query, Analysis and Visualization of Multi-Dimensional Relational Databases. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 8(1):52–65, 2002.

[140] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware Acceleration for Spatial Selections and Joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 455–466, 2003.

[141] Guo-Dao Sun, Ying-Cai Wu, Rong-Hua Liang, and Shi-Xia Liu. A Survey of Visual Analytics Techniques and Applications: State-of-the-Art Research and Future Challenges. *Journal of Computer Science and Technology (JCST)*, 28(5):852–867, 2013.

[142] Yufei Tao and Dimitris Papadias. Adaptive Index Structures. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2002.

[143] Yufei Tao, Dimitris Papadias, and Jun Zhang. Aggregate Processing of Planar Points. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, pages 682–700, 2002.

[144] Farhan Tauheed, Laurynas Biveinis, Thomas Heinis, Felix Schürmann, Henry Markram, and Anastasia Ailamaki. Accelerating Range Queries For Brain Simulations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 941–952, 2012.

[145] TLC Trip Record Data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml, 2015.

[146] Tiankai Tu and David R. O'Hallaron. A computational database system for generating unstructured hexahedral meshes with billions of elements, 2004.

[147] John Wilder Tukey. *Exploratory Data Analysis*. Pearson, 1977.

[148] Twitter API. https://dev.twitter.com/.

[149] Eleni Tzirita Zacharatou, Harish Doraiswamy, Anastasia Ailamaki, Cláudio T. Silva, and Juliana Freire. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *PVLDB*, 11(3):352–365, 2017.

[150] Eleni Tzirita Zacharatou, Farhan Tauheed, Thomas Heinis, and Anastasia Ailamaki. RUBIK: Efficient Threshold Queries on Massive Time Series. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2015.

[151] 10 Billion. https://www.uber.com/newsroom/10-billion/.

[152] Uber Hits 5 Billion Rides Milestone. https://www.uber.com/en-SG/blog/uber-hits-5-billion-rides-milestone/.

[153] Uber geofence. https://eng.uber.com/go-geofence/.

[154] Engineering Intelligence Through Data Visualization at Uber. https://eng.uber.com/data-viz-intel/.

[155] Thatcher Ulrich. Loose octrees. *Game Programming Gems*, 1:434–442, 2000.

[156] Frank van Diggelen and Per Enge. The world's first GPS MOOC and worldwide laboratory using smartphones. In *Proceedings of the International Technical Meeting of the Satellite Division of The Institute of Navigation (ION GNSS+)*, pages 361–369, 2015.

[157] Peter van Oösterom and Eric Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the International Symposium on Spatial Data Handling (SDH)*, pages 1016–1029, 1990.

[158] Ines Fernando Vega Lopez, Richard T. Snodgrass, and Bongki Moon. Spatiotemporal Aggregate Computation: A Survey. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 17(2):271–286, 2005.

[159] Maarten Vermeij, Wilko Quak, Martin Kersten, and Niels Nes. Monetdb, a novel spatial columnstore dbms. In *Academic Proceedings of the Free and Open Source for Geospatial (FOSS4G) Conference, OSGeo*, 2008.

[160] Lu Wang, Robert Christensen, Feifei Li, and Ke Yi. Spatial Online Sampling and Aggregation. *PVLDB*, 9(3):84–95, 2015.

[161] Sheng Wang, David Maier, and Beng Chin Ooi. Fast and Adaptive Indexing of Multidimensional Observational Data. *PVLDB*, 9(14):1683–1694, 2016.

[162] Andrew B. Watson et al. Temporal sensitivity. *Handbook of perception and human performance*, 1:6–1, 1986.

[163] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, pages 359–370, 1991.

[164] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 516–523, 1996.

[165] Hadley Wickham. Bin-summarise-smooth: a framework for visualising large data. Technical report, had.co.nz, 2013.

[166] Niels Willems, Huub Van De Wetering, and Jarke J. Van Wijk. Visualization of vessel movements. *Computer Graphics Forum (CGF)*, 28(3):959–966, 2009.

[167] K. Wu, S. Ahern, E. W. Bethel, J. Chen, H. Childs, E. Cormier-Michel, C. Geddes, J. Gu, H. Hagen, B. Hamann, W. Koegler, J. Lauret, J. Meredith, P. Messmer, E. Otoo, V. Perevoztchikov, A. Poskanzer, Prabhat, O. Rübel, A. Shoshani, A. Sim, K. Stockinger, G. Weber, and W. M. Zhang. FastBit: Interactively Searching Massive Data. *Journal of Physics: Conference Series*, 180(1), 2009.

[168] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing Bitmap Indices with Efficient Compression. *ACM Transactions on Database Systems (TODS)*, 31(1), 2006.

[169] Dong Xie, Feifei Li, Bin Yao, Gefei Li, Liang Zhou, and Minyi Guo. Simba: Efficient In-Memory Spatial Analytics. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1071–1085, 2016.

[170] Xiaopeng Xiong and Walid G. Aref. R-trees with update memos. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 22–22, 2006.

[171] Jia Yu, Jinxuan Wu, and Mohamed Sarwat. GeoSpark: a cluster computing framework for processing large-scale spatial data. In *Proceedings of the SIGSPATIAL International Conference on Advances in Geographic Information Systems*, volume 70, pages 1–4, 2015.

[172] Jianting Zhang and Simin You. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the ACM SIGSPATIAL International Workshop on analytics for Big Geospatial Data (BigSpatial)*, pages 23–32, 2012.

[173] Jianting Zhang, Simin You, and Le Gruenwald. High-performance online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. In *Proceedings of the International Workshop on Data Warehousing and OLAP (DOLAP)*, pages 89–96, 2012.

[174] Jianting Zhang, Simin You, and Le Gruenwald. High-Performance Spatial Join Processing on GPGPUs with Applications to Large-Scale Taxi Trip Data. Technical report, The City College of New York, 2012.

[175] Jianting Zhang, Simin You, and Le Gruenwald. Efficient Parallel Zonal Statistics on Large-Scale Global Biodiversity Data on GPUs. In *Proceedings of the ACM SIGSPATIAL International Workshop on analytics for Big Geospatial Data (BigSpatial)*, pages 35–44, 2015.

[176] Geraldo Zimbrao and Jano Moreira de Souza. A Raster Approximation For Processing of Spatial Joins. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 558–569, 1998.

Eleni Tzirita Zacharatou

Curriculum Vitae

Education

2013–2019 **Ph.D. in Computer Science**, *École Polytechnique Fédérale de Lausanne (EPFL)*.
Thesis Efficient Query Processing for Spatial and Temporal Data Exploration
Advisor Prof. Anastasia Ailamaki

2007–2013 **Diploma in Electrical & Computer Engineering**, *National Technical University of Athens (NTUA)*.
Thesis Automatic Music Transcription
Advisor Prof. Petros Maragos
GPA 9.06/10

Employment

09/2013–08/2019 **Research Assistant**, *EPFL*, Switzerland.
- Member of the Data-Intensive Applications and Systems (DIAS) laboratory
- Research on spatial and temporal data management for exploratory applications

07/2016-10/2016 **Visiting Scholar**, *NYU*, USA.
- Member of the Visualization, Imaging and Data Analysis Research Center (VIDA)
- Project: Real-Time Spatial Aggregation using GPU Rasterization
- Supervisors: Prof. Juliana Freire and Dr. Harish Doraiswamy

01/2013-08/2013 **Intern**, *EPFL*, Switzerland.
- Member of the Data-Intensive Applications and Systems (DIAS) laboratory
- Project: Efficient Time Series Bitmap Indexing through Quadtree-based Compression
- Supervisors: Prof. Anastasia Ailamaki and Prof. Thomas Heinis

Publications

Conferences

VLDB 2018 **E. Tzirita Zacharatou**, H. Doraiswamy, A. Ailamaki, C. Silva and J. Freire, GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons, 44^{th} *International Conference on Very Large Data Bases, Rio de Janeiro, Brazil, 2018*.

ICDE 2018 D. Sidlauskas, S. Chester, **E. Tzirita Zacharatou** and A. Ailamaki, Improving Spatial Data Processing by Clipping Minimum Bounding Boxes, 34^{th} *International Conference on Data Engineering, Paris, France, 2018*.

SSDBM 2015 **E. Tzirita Zacharatou**, F. Tauheed, T. Heinis and A. Ailamaki, RUBIK: Efficient Threshold Queries on Massive Time Series, 27^{th} *International Conference on Scientific and Statistical Database Management*, San Diego, California, USA, 2015.

EPFL IC IINFCOM DIAS BC 245 – Station 14, CH-1015 Lausanne
✉ *eleni.tzirita.zacharatou@gmail.com*
www.linkedin.com/in/eleni-tzirita-zacharatou

ECESCON 2013 **E. Tzirita Zacharatou** and P. Maragos, Signal Processing Methods for Automatic Transcription of Piano Music, 6^{th} *Annual Conference of Students of Electrical & Computer Engineering*, Athens, Greece, 2013.

Demonstrations

SIGMOD 2018 H. Doraiswamy, **E. Tzirita Zacharatou**, F. Miranda, M. Lage, A. Ailamaki, C. Silva and J. Freire, Interactive Visual Exploration of Spatio-Temporal Urban Data Sets using Urbane, *International Conference on Management of Data, Houston, Texas, USA, 2018.* **Best Demonstration Award**.

Workshops

ExploreDB 2016 M. Pavlovic, **E. Tzirita Zacharatou**, D. Sidlauskas, T. Heinis and A. Ailamaki, Space Odyssey - Efficient Exploration of Scientific Data, 3^{rd} *International Workshop on Exploratory Search in Databases and the Web, San Francisco, USA, 2016.*

Awards

2018 Best Demonstration Award, ACM SIGMOD Conference.

2016 ExploreDB Workshop (co-located with SIGMOD/PODS) Travel Award.

Conference Presentations and Invited Talks

07/2019 **GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons**, HDMS, Athens, Greece.

02/2019 **Interactive and Exploratory Spatio-Temporal Data Analytics**, DIMA Research Seminar, TU Berlin, Germany.

08/2018 **GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons**, VLDB, Rio de Janeiro, Brazil.

06/2018 **Interactive Exploration of Urban Data with GPUs**, Short Presentation, IC Research Day, EPFL, Switzerland.

04/2018 **Improving Spatial Data Processing by Clipping Minimum Bounding Boxes**, ICDE, Paris, France.

06/2017 **Real-Time Spatial Aggregation using GPU Rasterization**, EcoCloud Annual Event, Lausanne, Switzerland.

09/2016 **Indexing the brain**, Data Science Lunch Talk, NYU Center for Data Science, New York, USA.

06/2015 **RUBIK: Efficient Threshold Queries on Massive Time Series**, SSDBM, San Diego, USA.

Student Supervision

Fall 2018 Vlad Mihaescu (2nd year Master's at EPFL)

Project Evaluating range queries on interval and range encoded time series data with RUBIK

Summer 2018 Maria Ilina (Summer Intern at EPFL)

Project Efficient bitmap indexing via partitioning, reordering and compression

EPFL IC IINFCOM DIAS BC 245 – Station 14, CH-1015 Lausanne
✉ *eleni.tzirita.zacharatou@gmail.com*
www.linkedin.com/in/eleni-tzirita-zacharatou

Summer 2018	Parand Alizadeh (Summer Intern at EPFL)
Project	Devising a density-driven packing strategy for efficient spatial data access
Summer 2015	Chenxi Sun (Summer Intern at EPFL)
Project	Benchmarking Spatial Queries in PostGIS

Teaching Assistantships

2017, 2018	Database Systems
2014, 2018	Information, Computation, Communication
2017	Programming I (in C)
2016	Probability and Statistics
2015	Computer Architecture
2015	Introduction to Database Systems
2014	Introduction to Object Oriented Programming (in Java)

Professional Activities

2018	Program Committee Member, CIKM, Short Research Papers Track
2017	External Reviewer, ICDE, Research Track

In The News

2018	*Student-teacher Duo from EPFL Wins Best Demonstration Award at SIGMOD 2018*, Ecocloud News.
2018	*Tandon Team Takes Home Top Honors at Association for Computing Machinery Conference*, Tandon School of Engineering News.

Languages

Greek	Native proficiency
English	Full professional fluency
French	Full professional fluency
German	Elementary proficiency

Extracurricular

2010	Piano Teaching Diploma (Ptychio), Grade Excellent, Athenaeum Conservatory
2005	Certificate in Classical Harmony, Heraklion Conservatory
1996 - 2009	Studies in violin, Heraklion Conservatory & Athenaeum Conservatory
2001 - 2010	Participation in choirs and orchestral ensembles

EPFL IC IINFCOM DIAS BC 245 – Station 14, CH-1015 Lausanne
✉ *eleni.tzirita.zacharatou@gmail.com*
www.linkedin.com/in/eleni-tzirita-zacharatou

**Ce document a été imprimé au Centre d'impression EPFL,
imprimerie climatiquement neutre, certifiée myClimate.**