National Technical University of Athens

School of Electrical
and Computer Engineering

Computer Science Division

Adaptive Indexing for Interactive Visual Exploration and Analytics

Thesis

submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

by

Stavros Maroulis

Diploma in Electrical and Computer Engineering
National Technical University of Athens

Athens, July 2023

Εθνικό Μετσόβιο Πολυτεχνείο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων

Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Προσαρμοστικη ευρετηριαση για διαδραστικη οπτικη εξερευνηση και αναλυτικη

Διδακτορική Διατριβή

του

Σταύρου Μαρούλη

Διπλωματούχου Ηλεκτρολόγου Μηχανικού & Μηχανικού Υπολογιστών
Εθνικού Μετσοβίου Πολυτεχνείου

Αθήνα, Ιούλιος 2023

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Προσαρμοστική ευρετηρίαση για διαδραστική οπτική εξερεύνηση και αναλυτική

Διδακτορική Διατριβή

του

Σταύρου Μαρούλη

Διπλωματούχου Ηλεκτρολόγου Μηχανικού & Μηχανικού Υπολογιστών
Εθνικού Μετσοβίου Πολυτεχνείου

Συμβουλευτική Επιτροπή: Ι. Βασιλείου
Γ. Παπαστεφανάτος
Γ. Μέντζας

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή την 5η Ιουλίου 2023.

Ι. Βασιλείου
Ομότ. Καθ. ΕΜΠ

Γ. Παπαστεφανάτος
Ερευνητής Β'
Ε. Κ. ΑΘΗΝΑ

Γ. Μέντζας
Καθ. ΕΜΠ

Π. Βασιλειάδης
Καθ. Παν. Ιωαννίνων

Δ. Τσουμάκος
Αν. Καθ. ΕΜΠ

Ν. Κοζύρης
Καθ. ΕΜΠ

Γ. Στάμου
Καθ. ΕΜΠ

Αθήνα, Ιούλιος 2023

. . .

Σταύρος Μαρούλης
Διδάκτωρ Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

. . .

Stavros Maroulis
Doctor of Philosophy at the National Technical University of Athens

The approval of the doctoral dissertation by the School of Electrical and Computer Engineering of the National Technical University of Athens does not imply acceptance of the author's opinions.

Contents

List of Figures

List of Tables

PREFACE

This dissertation fulfills the requirements for the Doctor of Philosophy degree at the National Technical University of Athens' School of Electrical and Computer Engineering, Greece. The research was conducted at the Knowledge and Database Systems Laboratory of NTUA and the Institute for the Management of Information Systems at the ATHENA Research Center.

This work has been completed thanks to the assistance and encouragement of several individuals. First, I express my gratitude to Prof. Yannis Vasileiou, whose guidance proved instrumental in the realization of this work. I also extend my appreciation to Dr. George Papastefanatos, Dr. Nikos Bikakis, and Prof. Panos Vasileiadis. Their advice and contributions have been indispensable, and their passionate dedication to their research has served as my inspiration. I also owe a great deal of gratitude to all my colleagues at the Athena R.C. for their support and cooperation throughout this journey.

My heartfelt thanks go to my family and friends, who have been a pillar of support and patience throughout these years. Special acknowledgment is due to Evmorfia Biliri for our shared journey and friendship since our undergraduate years. I look forward to finding a similar acknowledgment in her dissertation.

Last but certainly not least, my deep appreciation and love to the one person who has endured even my worst moods when the pressures of this work became overwhelming.

Stavros Maroulis
Athens, July 2023

ΠΕΡΙΛΗΨΗ

Η παρούσα διατριβή παρουσιάζει νέες τεχνικές ευρετηρίασης που στοχεύουν στη διευκόλυνση της οπτικής εξερεύνησης δεδομένων αποθηκευμένων σε μεγάλα πρωτογενή αρχεία. Στη σύγχρονη εποχή, τα δεδομένα παράγονται με εξαιρετική ταχύτητα και σε τεράστιες ποσότητες, και η ικανότητα για γρήγορη επεξεργασία και κατανόηση αυτών των δεδομένων γίνεται ολοένα και πιο κρίσιμη. Τα συμβατικά εργαλεία εξερεύνησης δεδομένων βασίζονται σε μεγάλο βαθμό στα παραδοσιακά Συστήματα Διαχείρισης Βάσεων Δεδομένων (ΣΔΒΔ), τα οποία απαιτούν φόρτωση δεδομένων και ευρετηρίαση τους για μπορέσουν να αναλυθούν. Ωστόσο, αυτές οι διαδικασίες μπορούν να είναι ακριβές, χρονοβόρες και μη πρακτικές, ιδίως όταν τα δεδομένα ενδέχεται να μη χρησιμοποιηθούν ξανά μετά την ανάλυση τους.

Αρχικά εξετάζονται οι αδυναμίες των υπαρχόντων εργαλείων και μεθοδολογιών για την εξερεύνηση πρωτογενών δεδομένων, επισημαίνοντας την ανάγκη για ένα πιο αποτελεσματικό σύστημα. Στη συνέχεια, παρουσιάζεται ένα μοντέλο οπτικής εξερεύνησης όπου οι ενέργειες του χρήστη μεταφράζονται σε λειτουργίες πρόσβασης στα δεδομένα. Επιπλέον, εξετάζονται και παρουσιάζονται νέες τεχνικές ευρετηρίασης στη μνήμη, καθώς και μοντέλα κόστους, με ιδιαίτερη έμφαση στην προσαρμοστική ευρετηρίαση και τις δομές δεδομένων με ελαφρύτερο αποτύπωμα στη μνήμη. Αυτές οι τεχνικές είναι ειδικά σχεδιασμένες για τη διαχείριση μεγάλων όγκων πρωτογενών δεδομένων, ελαχιστοποιώντας αποτελεσματικά το κόστος πρόσβασης στο αρχείο δεδομένων και ξεκινώντας γρήγορα την αναλυτική εξερεύνηση του χρήστη, δημιουργώντας μια αρχική έκδοση του ευρετηρίου όταν ο χρήστης ζητά πρώτη φορά να αναλύσει ένα αρχείο. Αυτό το ευρετήριο γίνεται πιο λεπτομερές και προσαρμόζεται στην εξερεύνηση του χρήστη με κάθε ενέργεια του χρήστη. Επιπλέον, για την αντιμετώπιση σεναρίων με περιορισμένους υπολογιστικούς πόρους, εισάγεται ένας μηχανισμός αρχικοποίησης του ευρετηρίου που λαμβάνει υπόψιν τη διαθέσιμη μνήμη και προτείνονται αποτελεσματικοί αλγόριθμοι για την επίλυση του αντίστοιχου προβλήματος βελτιστοποίησης. Μέσω εκτενών πειραμάτων με πραγματικά και συνθετικά σύνολα δεδομένων, οι προτεινόμενες τεχνικές αποδεικνύονται ότι υπερτερούν των υπαρχόντων λύσεων, ανταποκρινόμενες έτσι στην ανάγκη για πιο αποτελεσματικές μεθόδους εξερεύνησης ακατέργαστων δεδομένων.

Αυτές οι τεχνικές ευρετηρίασης αποτελούν τη βάση του συστήματος RawVis, επιτρέποντας αποτελεσματική ανάλυση των δεδομένων, παρακάμπτοντας τα ακριβά στάδια προεπεξεργασίας τους, όπως η φόρτωση και η ευρετηρίαση τους σε ένα ΣΔΒΔ. Το RawVis παρέχει μια πλήρη και αποτελεσματική αρχιτεκτονική πελάτη-διακομιστή για οπτική εξερεύνηση δεδομένων απευθείας από τα πρωτογενή αρχεία, περιλαμβάνοντας μια πλούσια διεπαφή χρήστη που παρουσιάζει μια ευρεία γκάμα επιλογών για οπτικοποίηση και ανάλυση. Μέσω μιας εκτενούς μελέτης χρηστών, αποδεικνύεται η ικανότητα του συστήματος να προσφέρει οπτική ανάλυση μεγάλων αρχείων πρωτο-

γενών δεδομένων.

Συνοψίζοντας, αυτή η διατριβή προσφέρει μια σημαντική συνεισφορά στον τομέα της αναλυτικής δεδομένων, παρουσιάζοντας ένα νέο σύστημα και τεχνικές που βελτιώνουν σημαντικά την αποδοτικότητα της διαχείρισης των δεδομένων, μειώνουν τη χρήση πόρων και ενισχύουν την εμπειρία του χρήστη σε ό,τι αφορά την ταχύτητα και την αλληλεπίδραση.

ABSTRACT

This thesis introduces novel indexing techniques aimed at facilitating the visual exploration of data stored in large raw files. In today's data-driven society, data is produced at an extraordinary pace, and the ability to rapidly process and comprehend this data is becoming increasingly vital. Conventional data exploration tools heavily rely on Database Management Systems (DBMS), which require data loading and indexing for analysis. However, these procedures can be expensive, time-consuming, and impractical, especially when the data may be discarded after analysis.

The initial part of this thesis sheds light on the shortcomings of existing tools and methodologies for in-situ data exploration, establishing a compelling argument for a more efficient system. Subsequently, we present a formal visual exploration model where user operations are translated into data access operations. Furthermore, we unveil novel memory indexing techniques and cost models, with a special emphasis on adaptive indexing and lightweight data structures. These techniques are specifically designed to manage large volumes of raw data, effectively minimizing the I/O cost of accessing the data file and quickly initiating user exploratory analysis by generating a crude version of the index when the user first requests to analyze a file. This index becomes more detailed and adapts to user exploration with each user operation. Additionally, to handle scenarios with limited resources, a resource-aware index initialization mechanism is introduced, and efficient approximation algorithms are proposed to solve the corresponding optimization problem. Through extensive experimentation using both real and synthetic datasets, the proposed techniques have been demonstrated to outperform existing solutions, thus addressing the need for more efficient and intuitive raw data exploration methods.

These indexing techniques and schemes form the backbone of the RawVis system, enabling efficient query processing and bypassing expensive data preprocessing steps such as data loading and DBMS indexing. RawVis provides a complete and efficient client-server architecture for visual data exploration directly over the raw data files, including a rich user interface that presents a wide array of options for visualization and analysis. The application of RawVis is demonstrated through a user study, highlighting its ability to offer immediate and meaningful analytics.

In summary, this thesis offers a significant contribution to the field of raw data exploration by unveiling a novel system and techniques that notably enhance data handling efficiency, reduce resource usage, and amplify the user experience in terms of speed and interactivity.

Chapter 1

Introduction

1.1 Background and Motivation

Recent advancements in open science practices have led to the proliferation of a vast number of datasets. These are openly shared in accessible repositories, and a significant portion is in the form of raw data, i.e., files in standard formats such as .csv and .json. Consumers of such datasets (e.g. scientists) usually have limited skills in complex data management and analysis, and may have *limited resources* or *commodity hardware* for use (e.g., scientist's laptop), in contrast to, e.g., a distributed environment. At the same time the tasks the users wish to accomplish are fairly typical and involve having a quick overview and then exploring and analyzing the contents of a big raw data file preferably by easy-to-use visual ways, such as 2D visualization techniques (e.g., scatter plot, map) and simple plots (e.g., bar charts), avoiding the tedious tasks of data loading, preparation and indexing.

One common task in data exploration scenarios involves *in-situ* visual data analysis, in which data scientists wish to *visually interact and analyze* large (and *dynamic*) raw data files (e.g., CSV). In such scenarios, users need to perform the analysis directly over the raw files, avoiding the tedious tasks of loading and indexing the data in a data management system. Still, they expect a very small *data-to-analysis time* and they wish to interact via a rich set of *visual exploration and analytic operations.*

To this end, efficient in-situ processing and analysis of data stored in raw files is a major challenge for a large number of real-world tasks over diverse domains, such as astronomy, business intelligence, finance, telco, etc.

To better illustrate the scenario we are discussing, consider the following examples from the fields of astronomy and telecommunication:

Astronomy Example. Consider a scientist (e.g., astronomer) who wishes to visually explore and analyze sky observations stored in raw data files (e.g., csv) using available datasets; e.g., Sloan Digital Sky Survey (SDSS)[1], Palomar Transient Factory[2], Zwicky Transient Facility[3], Large Synoptic Survey Telescope[4], in which hundreds of millions of sky objects (e.g., stars) are described.

First, the scientist selects the file and visualizes a part of it using scatter plots with the sky coordinates (e.g., right ascension and declination) [20]. Then, they

[1] www.sdss.org

[2] www.ptf.caltech.edu/iptf

[3] www.ptf.caltech.edu/ztf

[4] www.lsst.org

may focus on a sky region (e.g., defining coordinates and area size), for which all contained sky objects are *rendered*; *move* (e.g., pan left) the visualized region in order to explore a nearby area; or *zoom-in/zoom-out* to explore a part of the region or a larger area, respectively.

They may also click on a single or a set of sky objects and view *details*, such as *name* and *diameter*; *filter* out objects based on a specific characteristic, e.g., *diameter* larger than 50 km; or *analyze* data considering all the points in the visualized region, e.g., compute the average *age* of the visualized objects.

Further, they may also wish to perform further visual analysis of the sky objects in the area they currently explore by generating for example a bar chart to visualize the number of sky objects per type, or a heatmap to visualize the number of objects per type and observation program.

Telecommunication Example. The data scientists working in telco companies analyze network data in order to get insights regarding the network quality. Such data are commonly stored in large comma-separated data files and contain signal and latency measurements crowdsourced from IoT mobile devices, e.g., connected cars, mobile phones.[5]

Assume that a data scientist wishes to *visually explore* the network data using a map. First, the user *renders* on the map the signal measurements located in a specific geographic area, views *details* (e.g., network provider) for the points visualized, or $filters$ out the ones that refer to a specific provider. Next, they may *move* (e.g., pan left) the visualized region in order to explore a nearby area; or *zoom-in/out* to explore a part of the region or a larger area, respectively. The scientist is also interested in *analyzing* the data considering the points in the visualized region by computing *statistics between numeric attributes*, e.g., the Pearson correlation coefficient between the signal strength and the bandwidth; or *visualize* its values using a *scatter plot*. Finally, the user may also be interested to *visually analyze data*, exploiting also the crucial information included in the *categorical attributes*; e.g., via a *heatmap* to present the average signal strength per provider and network technology, or a *bar chart* to present the average signal strength for each provider, or a *parallel coordinates* chart to present the number of measures grouped by provider, brand, and network technology.

1.2 The Challenges of Efficient In-Situ Visual Exploration and Analytics

As highlighted, the significance of visual exploration and analytics applied to raw data is fundamental in numerous real-world applications. Users often wish to utilize a diverse range of visualization techniques that allow them to interact with data within a 2D visual environment. For instance, rendering data objects on a map and engaging in interactive explorations of the area. Users can then concentrate their analysis on a specific region by zooming in, or alternatively, they can zoom out to analyze a larger area and draw more generalized conclusions.

Complementing these map-based interactions, other visualization constructs such as bar charts and heatmaps prove to be indispensable tools for effective data analy-

[5] For example, https://www.tutela.com

sis. These graphical representations are widely applied across various data analysis tasks, ranging from feature extraction to OLAP (Online Analytical Processing) analysis, regression, and comparative analysis of spatial data [68].

Beyond the above-mentioned operations and visualization techniques, filter operations provide the backbone for *effective exploration mechanisms*, such as *faceted search*. This technique allows users to delve into specific aspects of their data, refining their exploration and analysis journey. These types of analysis and queries have been widely optimized in traditional data warehouse systems, via spatial and multidimensional indexes, or materialized aggregated views. However, these methods require loading the data, as well as constructing and tuning the indexes that would be necessary for improving the interactivity of such visual analysis.

In-situ techniques, on the contrary, attempt to avoid the overhead of moving, loading and indexing the data in a DBMS. The key objective is how to offer fast user interactions without a long preprocessing phase.

The major challenges of such exploration scenarios include:

- First, *how can we support a non-expert user with limited programming or scripting skills to access and analyze raw data from a file through visual ways*, i.e., via an intuitive set of visual rather than data-access (e.g., querying) operations, without being overwhelmed with any data pre-prossessing tasks, such as extracting, loading and indexing data to a database?

- Next, *how can we keep the response time of such visual operations significantly small* (e.g., less than 1sec) in order to be acceptable by the user?

- Finally, *how can we perform the aforementioned operations in machines with limited computational, memory and space resources*, i.e., using commodity hardware?

Most experimental and commercial visualization tools perform well for ad-hoc visualizations of *small files* (e.g., showing a trend-line or a bar chart) or over aggregated data (e.g., summaries of data points, into which users can zoom in), which *can fit in main memory*. For *larger files*, the tools usually require a *preprocessing step* for data to be *loaded*[6], *indexed* (e.g., a spatial index like R-tree) and handled either via a traditional database or a distributed storage hosted in a *non-commodity hardware*. Further, many commercial RDBMs and visualization tools offer also capabilities for querying external raw data files (e.g., external tables)[7]; however, they limit themselves to recurrent file access each time a query is performed and achieve poor performance [7], prohibitive for the interactive exploration purposes.

In recent years, several *adaptive indexing techniques* have emerged with the goal of avoiding upfront construction of a complete index. These techniques aim at minimizing the cost associated with indexing and enable self-organizing DBMSs that require less human administration. In such approaches, the indexes are incrementally adjusted and/or the physical order of data is refined during query processing, in accordance with the characteristics of the workload. [51, 42, 99, 46, 71, 42, 43, 70, 75].

[6] For example, Tableau has limitations on the size of the data file that can be loaded for visualization [5].

[7] For example, Oracle [2], MySQL [1] and PostgresSQL [3] provide mechanism that enable SQL querying of csv files.

However, in most cases the data has to be previously loaded in the system, i.e., a preprocessing phase is considered.

On the contrary, the *in-situ paradigm* has been recently adopted when analysis should be performed directly on raw data files (e.g., CSV, JSON), avoiding the overhead of fully loading and indexing the data in a DBMS. Similarly, to traditional in-database adaptive methods, in-situ techniques achieve performance by building indexes on-the-fly and progressively readjusting them as the user explores data. Works in this area have proposed techniques for progressive loading and/or indexing of raw data, for "generic" in-situ query processing [7, 49, 45, 90, 57, 56, 73, 74].

Most of these works, however, study the generic in-situ querying problem without focusing on the specific needs for supporting efficient and interactive raw data visualization and exploration. Hence, the challenge in these scenarios is to achieve optimization of such analysis, ensuring that visual interaction with raw data is performed efficiently on very large input files using commodity hardware.

1.3 Research Objectives

This research is motivated by the increasing need for efficient and effective in-situ visual data analysis in various domains. It aims to address the challenges associated with in-situ processing and interactive visual analysis of large, raw data files, and contribute to the development of new techniques and tools that can support data scientists in their work.

The following objectives outline a set of requirements for efficient in-situ visual exploration and analytics, shaping the direction of the research presented in this thesis:

- **Minimize Data-to-Analysis Time:** The aim is to enable quick data analysis without lengthy pre-processing. To this end, we need to minimize the cost for parsing the data files. Any indexing should be performed *on-the-fly* to ensure *minimal data-to-analysis time*. Even for large datasets, parsing and index creation should be brief.

- **Efficient Evaluation of User Operations:** Quick evaluation of exploratory and analytical operations on raw files during in-situ exploration is vital. For large datasets, it's essential to use an index to enable fast operation evaluation.

- **Minimizing I/O Operations:** Given the significant impact of I/O operations on response time, an essential requirement is to limit the number of objects read from the file during query processing.

- **Optimizing Memory Usage:** With large raw data files and in-memory indexes exceeding available memory on standard hardware, a key objective is identifying the optimal data subset for indexing.

- **Adapting to User Interaction:** Efficient query evaluation is crucial in exploration scenarios. Given the prohibitive cost of full upfront indexing in the in-situ setting and considering the requirements above, it's necessary to adapt the index to user interaction. This can bypass the need for a fully detailed index, making the process more efficient.

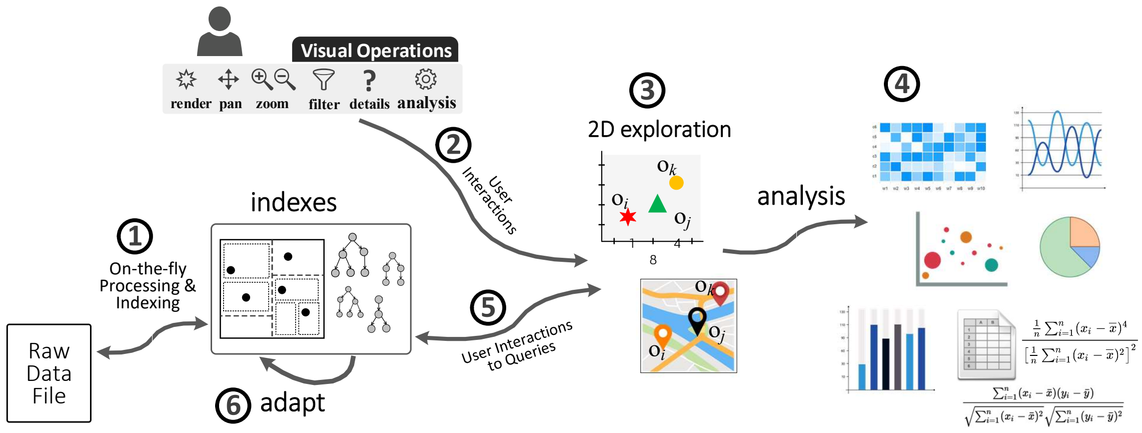


Figure 1.1: Visual Exploration Scenario Overview

These requirements underscore the need for a solution that addresses the distinct challenges of in-situ visual exploration and analytics. They set the foundation for the solution we will propose and explore in the subsequent sections of this thesis.

1.4 In-Situ Visual Exploration: Our Working Scenario

This thesis introduces innovative methods for exploratory visual analysis, enabling direct utilization of data from raw data files and thereby circumventing the traditional requirement for a database management system (DBMS). This approach's overarching aim are outlined using a hypothetical scenario, presented in Figure 1.1.

In this scenario, imagine a user that wishes to visually explore data through a 2D visualization technique, such as a scatter plot or map, and perform further analysis using various chart-based visual analytics techniques, as well as statistical methods. The priority on 2D visual exploration in this scenario is purposeful, serving to support one of the most prevalent methods of visual analysis. While working in more than two dimensions or broader query classes is possible, both the 2.05-dimensional nature of the human eye [93] and the 2-dimensional nature of the media (being paper or screen) make the key two-dimensional operators, like the aforementioned ones, being *fundamental*, especially, for the initial part of the knowledge extraction process, which is data exploration. Many datasets naturally align with this form of analysis as they contain positional information (e.g., geographic coordinates or astronomical coordinates like right ascension and declination). Additionally, the 2D characteristics inherent to these exploration scenarios add an extra layer of complexity in contrast to one-dimensional scenarios (e.g., time series data), making them especially challenging to address.

The targeted visual analysis and its underlying principles are executed in a sequence of stages, detailed as follows:

① The user first selects the input file and a map as the underlying visualization layout. The file is parsed on-the-fly and an initial "crude" version of the index is constructed. ② Then, the user interacts and performs visual and analytic operations on the map ③. For example, they may generate visual data representations (e.g., bar charts, heatmaps), or use statistical approaches (e.g., Pearson correlation) ④.

Eventually, each user interaction and analytical operation is mapped to a query evaluated over the index structures ⑤, and triggers the readjustment of the index structure and the update of its contents ⑥.

1.5 Thesis Outline

To address the research objectives and challenges previously presented, this thesis introduces novel indexing structures and techniques to enable efficient and effective in-situ visual data analysis. The following chapters delve into these techniques, offering a detailed and comprehensive understanding of their design, implementation, and evaluation.

The structure of this work is as follows:

Chapter 2 provides a review of the existing literature, focusing on efficient indexing and querying of raw data, and efficient exploration and visualization techniques. Chapter 3 then introduces the basic concepts of the visual exploration model, forming the foundation of this thesis.

The discussion extends in Chapter 4 where we present VALINOR, a main-memory index for 2D in-situ visual exploration. In Chapter 5, we build upon this by introducing the VETI index, an extension of VALINOR that is optimized to handle categorical attributes.

Chapter 6 presents an application of these principles and techniques in a prototype system, RawVis, designed to facilitate the real-time visual exploration of raw data files.

Finally, Chapter 7 provides a conclusion to the research, revisiting the addressed challenges, and reflecting on the potential for future work in the area. The aim of this thesis is to provide valuable insights into the world of in-situ visual data analysis and pave the way for further research in this field.

Chapter 2

Related Work

2.1 Introduction

Several areas relate to the general problem of exploration and visualization of raw data, which can be grouped into two main categories: efficient indexing and querying of the raw data; and efficient and effective exploration and visualization techniques. On the indexing and query processing part, the most relevant one deals with in-situ query processing, i.e., how the time-consuming task of loading and indexing of the data can be avoided such that the time-to-analysis is minimized. Also, considering the focus given in this thesis on 2D exploration settings, we review several spatial indexes that have been proposed for 2D querying and analysis in database settings. On the visualization and exploration part, there is number of visualization-driven indexes, most of them operating in main memory, that aim at speeding up user exploration actions. In this section, we present in details these works and provide a comparison of our approach to them.

2.2 In-situ Query Processing

Data loading and indexing usually take a large part of the overall time-to-analysis for both traditional RDBMs and Big Data systems [45]. In-situ query processing aims at avoiding data loading in a DBMS by accessing and operating directly over raw data files. NoDB [7] is a philosophy for constructing a no-dbms querying engine over raw data, and PostgresRAW is one of the first efforts for in-situ query processing. PostgresRAW incrementally builds on-the-fly auxiliary indexing structures called “positional maps” which store the file positions of data attributes, as well as it stores previously accessed data into cache.

DiNoDB [90] is a distributed version of PostgresRAW. In the same direction, PGR [57] extends the positional maps in order to both index and query files in formats other than CSV. In the same context, Proteus [56] supports various data models and formats. Also, Slalom [73, 74] exploits the positional maps and integrates partitioning techniques that take into account user access patterns.

Raw data access methods have been also employed for the analysis of scientific data, usually stored in array-based files. In this context, Data Vaults [50] and SDS/Q [19] rely on DBMS technologies to perform analysis over scientific array-based file formats. Further, SCANRAW [24] considers parallel techniques to speed up CPU intensive processing tasks associated with raw data accesses.

To note that, several well-known DBMS support SQL querying over CSV files. Specifically, MySQL provides the CSV Storage Engine [1], Oracle offers the External Tables [2] and Postgres has the Foreign Data [3] However, these tools do not focus on user interaction, parsing the entire file for each posed query, and resulting in significantly low query performance [7] for interactive scenarios.

Despite their contributions, the aforementioned works primarily focus on the generic in-situ querying problem, without fully considering the specific needs for raw data visualization and exploration. For instance, the positional map in PostgresRAW is primarily used to reduce parsing and tokenization costs during query evaluation. However, it does not minimize the number of objects examined during a query's evaluation.

In the context of in-situ visual exploration, it is crucial to minimize the time taken to process user operations and update visual representations displayed to the user. The aim is not solely to reduce file parsing cost but also to decrease the number of data objects examined. In addition to indexing data objects based on their attribute values, we can enhance the efficiency of queries that involve calculating aggregates and statistics. This can be achieved by reusing previously computed statistics, thereby eliminating the need to access the file for subsets of the data.

Given the potentially very large size of raw data files, existing solutions also fall short as they overlook the consideration of available memory resources. By explicitly accounting for the accessible memory, and for instance, establishing a more granular indexing structure around the user's areas of exploration, we can optimize resource distribution, thus accelerating the early stages of exploration.

In contrast to existing approaches, this thesis places emphasis on the specific requirements and challenges of interactive visual exploration and analytics in the in-situ context. The solutions proposed herein aim to optimize user exploratory operations, such as pan and zoom in a 2D exploration scenario, as well as various chart-based visualizations and aggregate analytics. These proposals ensure efficient visual interaction with raw data on large input files, even on commodity hardware with limited memory resources, thereby underscoring the significance of taking memory requirements into consideration.

2.3 Exploratory Data Analysis

A core objective of this work is to address the challenges of providing effective and efficient exploratory data analysis techniques. Data exploration sessions usually start by employing statistical analysis to gain an overview of the various characteristics of the data and find underlying trends in an iterative process, where each exploratory query helps formulate the next one. Most traditional database systems provide support for basic statistical analysis (e.g., aggregates, top-k, etc). More advanced exploratory statistical analysis can be performed by tools like the R programming language [78] or NumPy and SciPy [4]. These tools cannot handle our in-situ exploratory scenario, though, since they either assume that the data fits in memory, or are integrated with traditional database systems which require a preprocessing phase.

One significant challenge in exploratory statistical analysis is the time required to compute statistics on large datasets, which often proves unsuitable for interactive settings. To address this issue and enable the reuse of computations in future queries, various approaches have been proposed. In [94], Data Canopy is introduced as a method to reduce the amount of data accessed during statistics calculation. It achieves this by synthesizing statistics from basic aggregates computed over chunks of data columns, which are then cached for reuse by subsequent queries.

However, while Data Canopy enables the reuse of cached basic aggregates for efficient calculation of more general statistics, it does not explicitly tackle the problem of fast exploration over large raw datasets. Furthermore, Data Canopy's focus is primarily on the one-dimensional setting, with its chunks defined over consecutive data items from a single column. Although useful, it does not cater to the specific requirements of common visual exploration settings involving two dimensions. Consequently, the chunking approach used in Data Canopy facilitates computing statistics over query ranges defined between two positions in a column set, but it cannot be effectively exploited for evaluating two-dimensional window queries.

2.4 Adaptive Indexing

An important research area related to the objectives of this thesis revolves around the concept of adaptive indexing. The basic idea of approaches like database cracking and adaptive indexing [51, 42, 99, 46, 71, 42, 43, 70, 75, 8, 81, 76, 47, 48, 37, 39, 82, 44] is to incrementally build and adapt indexes and/or refine the physical order of data, during query processing, following the characteristics of the workload.

However, in these works the data has to be previously loaded in the system, i.e., a preprocessing phase is required. As a result, these approaches are not suitable for in-situ query scenarios, where the cost of the preprocessing phase has to minimized. Additionally, the aforementioned works refine the (physical) order of data, performing highly expensive data duplication and allocate large amount of memory resources. As a result, these approaches are not suitable for in-situ query scenarios, where the cost of the preprocessing phase has to be minimized. Nevertheless, in the *in-situ scenarios* the analysis is performed directly over immutable raw data files considering limited resources. In addition, the existing cracking and adaptive indexing methods have been developed in the context of column-stores [39, 37, 46, 48, 47, 76, 8], or MapReduce systems [81].

2.5 Multi-dimensional Indexing

The focus of this work centers on an in situ exploration scenario, where users actively participate in interactive exploration utilizing two-dimensional visualizations, such as maps or scatter plots. This exploration involves various user operations, including panning and zooming, to navigate and examine the data. To enable efficient query processing across multiple dimensions, a diverse array of multi-dimensional index structures has been introduced in both traditional databases and Big Data systems.

Traditional spatial indexes, such as the R-tree, kd-tree, quadtree [34], are designed to improve the evaluation of range or nearest-neighbor queries on multi-dimensional data, and are widely available in both disk-based and main memory implementations. In R-trees [65], nearby objects are grouped together using minimum bounding rectangles, with rectangles at higher levels of the tree aggregating an increasing number of objects and leaf nodes containing the actual objects. In the same context, several variants have been proposed to solve some of its disadvantages. For example, X-trees [14] try to avoid the overlap in the bounding boxes, a common problem in higher dimensions, by introducing a splitting algorithm and the concept of supernodes. M-trees [25], another R-tree variant, are constructed using a distance metric and rely on the triangle inequality for more efficient range and k-nearest neighbor queries. In contrast to X-trees, M-trees suffer from large overlap.

R-trees, as well as its variants [65], consider several criteria (e.g., tree balance, space coverage, node overlaps, fill guarantees) in order to improve query processing.

As a result, even main memory implementations require substantial memory and time resources to construct, which makes them inappropriate for enabling the users to quickly start exploring and interacting with the data, as in the case of in situ data exploration. In contrast, lightweight indexing structures, such as those proposed in this thesis, are more favorable for in situ data exploration. These lightweight structures aim to expedite the raw data-to-visualization time and provide users with a simple set of 2D visual operations, prioritizing lower memory requirements and faster initialization time over more advanced aspects of spatial data management.

The indexing structures proposed in this work exhibit similarities with the grid file [72] as they both partition the space and organize data objects into tiles/cells. However, significant differences arise when considering the merging and splitting phases, methods, and criteria employed in each approach.

Firstly, in grid files, the merging and splitting phases occur during the grid construction phase. In contrast, our proposed indexes adapt by performing merging and splitting operations after construction, during runtime. Furthermore, these merging and splitting operations are executed incrementally and adaptively, based on user interactions.

Secondly, the criteria used to determine the merging and splitting differ between the two approaches. Grid files utilize criteria such as storage utilization, minimum/maximum number of objects per tile, number of I/O accesses, and budget size. In contrast, this work employs merging and splitting criteria based on user interactions, particularly queries. Additionally, this work estimates the initial structure characteristics, such as tile size, based on the first user's interaction. Furthermore, we compute and leverage specific metadata aimed at enhancing visual-based operations and analytics.

Building upon our review of existing multi-dimensional indexing structures, it is worth examining related studies that focus on distributed caching and performance improvement in multi-dimensional scenarios. [102] studies the problem of distributed caching of multi-dimensional raw arrays. The system implements a distributed caching system that improves the performance of queries that use frequently accessed data values, focusing on similarity join over arrays queries [101]. To this end, a method that selects which part of the data to be cached is proposed. This method is based on an R-tree which is incrementally enriched with the data that are accessed. Further, the caching mechanism, uses an algorithm to select in which node the cached data have to be stored in order to minimize data transfer. This algorithm is implemented as a search greedy algorithm which is based on incremental array view maintenance [103]. [102] considers raw data files, as well as the incremental indexing paradigm. However, it is important to note that these works investigate different settings and pursue distinct goals compared to the focus of this thesis. The primary objective of this work is to enable in-situ visual exploration of large raw data files using commodity hardware, without incorporating a distributed setting. Moreover, the fundamental goal of the proposed indexing structures and techniques in this work is to minimize I/O operations, enabling efficient and interactive evaluation of user operations throughout the exploration process.

2.6 Visual Exploration

Visual data exploration offers the users the ability to interact with the underlying data through visual ways, i.e., mapping user operations to data access and querying methods [77, 15, 9, 35]. In this context, the first efforts focused on developing visual querying languages for DBs such as [104, 10, 22, 23, 66]. Most of these languages address the need to offer the database analyst a visual way for syntactically expressing a query, rather than offering visual operations for interactive data exploration. In most interactive visualization systems, visual user operations (e.g., map panning) are used for specifying the actual query logic and several visualization languages have been proposed to to simplify the generation of such visualizations [28, 40, 41, 95].

2.6.1 Progressive Visualization

As already highlighted, a core challenge in visual exploration is *interactivity.* Many of the existing studies, based on human cognitive constraints, state that *a delay of one second* is the (most common) upper bound in interactive applications [63, 83, 21, 84]. To support interactivity, many systems adopt the *progressive paradigm* attempting to reduce the response time. [32, 11, 80, 100, 6, 33]. Progressive approaches, instead of performing all the computations in one step (that can take a long time to complete), splits them in a series of short chunks of approximate computations that improved with time. Therefore, instead of waiting for an unbounded amount of time, users can see the results unfolding progressively. These approaches can adjust the relation between the response time and the approximation error bounds.

2.6.2 Visualization Recommendation

In order to assist users throughout the visual exploration process, several approaches have been developed in the context of visualization recommendation [91]. These approaches recommend the most “suitable” visualizations by taking into account several factors, such as data characteristics, environment setting and available resources (e.g., screen resolution/size, available memory) [52, 18], user preferences and behavior [69, 36], examined task, etc. Especially considering data characteristics, there are several systems that recommend the most suitable visualization technique (and parameters) based on the type, attributes, distribution, or cardinality of the input data [58, 30, 64].

2.7 Data Structures & Indexing for Visual Data Exploration

Several data visualization systems have been created utilizing data structures and indexes specifically tailored for visual exploration, aiming to enhance efficiency and scalability. VisTrees [31] and HETree [18] are tree-based main-memory indexes that address visual exploration use cases, i.e., they offer exploration-oriented features such as incremental index construction and adaptation. Compared to our work, both indexes focus on one-dimensional visualization techniques (e.g., histograms), do not support categorical attributes and group-by analytics, and do not consider disk storage, i.e., data stored in-memory.

Nanocubes [61], Hashedcubes [27], SmartCube [62], Gaussian Cubes [92], and TopKubes [67] are main-memory data structures defined over spatial, categorical and temporal data. The aforementioned works are based on main-memory variations of a data cube in order to reduce the time needed to generate visualizations. Nanocubes [61] attempts to reduce the memory of the data cube by sharing nodes in a single tree structure. Hashedcubes [27] follows a different approach where, instead of materializing all possible aggregations, it uses a partial ordering of the dimensions and the notion of pivot arrays to calculate on-the-fly the aggregations missing. Smartcube [62] is a variation of Nanocubes, where instead of pre-computing all cuboids from the start, it chooses some important ones based on the user queries, in order to reduce memory usage. Also, it may adaptively change stored cuboids when querying patterns change. The indexes in the aforementioned works are generated during a preprocessing phase, and thus cannot be used in in-situ scenarios, e.g., they do not address problems related to reducing the initialization time. Moreover, these works assume that all the aggregations are materialized and stored in main memory, and can often require prohibitive amounts of memory.

Further, graphVizdb [17, 16] is a graph-based visualization tool, which employs a 2D spatial index (e.g., R-tree) and maps user interactions into window 2D queries. To support the operation of the tool, a partition-based graph drawing approach is proposed. Compared to the approaches presented in this work, graphVizdb requires a loading phase where data is first stored and indexed in a relational database system. In addition, it targets only graph-based visualization and interaction, whereas our approach offers interaction in 2D layouts, such as maps or scatter plots.

Spatial 2D indexing is also adopted in Kyrix [87]. Kyrix is a generic platform that supports efficient Zoom and Pan operations over arbitrary data types. Initially, the data is stored in a database and indexed using R-trees. In both graphVizdb and Kyrix the zoom levels are predefined, with each level having its own table and R-tree. Each Pan and Zoom operation is mapped to a rectangle 2D query, and based on the zoom level, is evaluated over the corresponding table and R-tree.

Compared to our setting, the aforementioned systems require a preprocessing phase where data is first stored and indexed in a database system. On the other hand, this thesis focuses on the in-situ setting, over limited memory resources. To improve query evaluation performance and reduce the I/O costs, the methods proposed here are based on in-memory incremental and adaptive indexing and query evaluation methods. On the other hand, in Kyrix, queries are evaluated over fixed database indexes, while a caching and prefetching strategy is used to reduce the database access cost.

Another difference is related to the evaluation of statistics. This work focuses on efficient statistics computations by utilizing stored metadata to reduce the required I/O operations. On the other hand, the aforementioned works do not study the problem of efficient statistics computations.

In another context, tile-based structures are used in visual exploration scenarios. Semantic Windows [54] considers the problem of finding rectangular regions (i.e., tiles) with specific aggregate properties in an interactive data exploration scenario. This work uses several techniques (e.g., sampling, adaptive prefetching, data placement) in order to offer interactive online performance. ForeCache [12] considers a client-server architecture in which the user visually explores data from a DBMS. The approach proposes a middle layer which prefetches tiles of data based on user interaction. Prefetching is performed based on strategies that predict next user's movements. This work considers different problems compared to the aforementioned approaches, but some of these methods can be exploited in our framework to further improve efficiency and estimate several parameters (e.g., splitting criteria, eviction and initialization policy).

2.8 Summary

This chapter provided an overview of existing works related to the data exploration and visualization of raw data. It discussed efficient indexing and query processing of raw data, exploratory data analysis, adaptive indexing, multi-dimensional indexing, and visual exploration. The chapter highlighted the limitations of existing approaches and set the foundation for the proposed solutions that aim to optimize user exploratory operations in the context of interactive in-situ visual exploration.

Chapter 3

Visual Exploration Model

3.1 Introduction

This chapter delves into the description of a formal visual exploration model that serves as the foundation of the indexing structures and techniques proposed in this thesis. The aim is to provide a comprehensive understanding of the model and its underlying principles, enabling the facilitation of exploratory visual analysis of data from raw data files.

The chapter first introduces the basic concepts of the visual exploration model. It defines some preliminaries regarding the structure of a raw data file and the objects within it, as well as the various types of attributes and how they are used in the visual exploration model. The model outlines the various operations performed by the user as they visually explore and analyze the raw data. Furthermore, the chapter describes the mapping between visual operations and exploratory queries. This mapping helps establish a clear connection between user interactions and the underlying data access and processing operations.

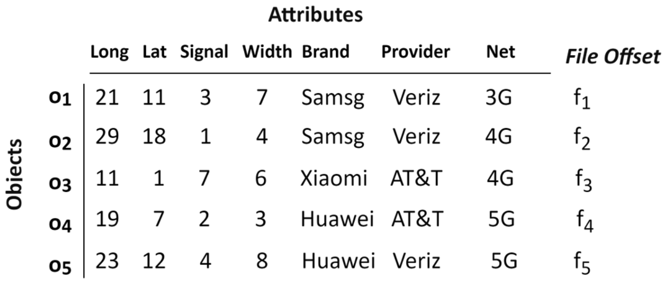
Overall, this chapter serves as an essential foundation for the subsequent discussions and presentation of the novel index structures and techniques proposed to enhance the performance and efficacy of in-situ visual exploratory analysis.

3.2 Basic Concepts

In this section, we define the foundational concepts pertinent to the visual exploration scenario described above.

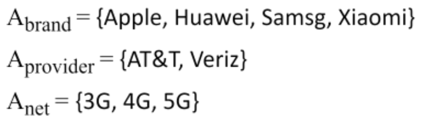
Raw Data File & Objects. We assume a *raw data file* $\mathcal{F}$ containing a set of *d-dimensional objects* $\mathcal{O}$. Each dimension j corresponds to an *attribute* $A_j \in \mathcal{A}$, where each attribute may be numeric or textual.

Each object o_i contains a list of d attributes $o_i = (a_{i,1}, a_{i,2}, ..., a_{i,d})$, and it is associated with an *offset* f_i (a hex value) pointing to the "position" of its first attribute (i.e., $a_{i,1}$) from the beginning of the file $\mathcal{F}$. Note that object entries can be either fixed or variable-length; in the latter case they are separated by a special-character; e.g., CR for a text file, that precedes the offsets. Note also, that we consider flat files, i.e., files containing objects that neither exhibit any nesting or any other complex structure (e.g., JSON formats), nor refer to data located in other files.



Objects	Attributes							File Offset
	Long	Lat	Signal	Width	Brand	Provider	Net	
o1	21	11	3	7	Samsg	Veriz	3G	f_1
o2	29	18	1	4	Samsg	Veriz	4G	f_2
o3	11	1	7	6	Xiaomi	AT&T	4G	f_3
o4	19	7	2	3	Huawei	AT&T	5G	f_4
o5	23	12	4	8	Huawei	Veriz	5G	f_5

(a) Raw Data File Sample



A_{brand} = {Apple, Huawei, Samsg, Xiaomi}

A_{provider} = {AT&T, Veriz}

A_{net} = {3G, 4G, 5G}

(b) Categorical Attributes Domains

Figure 3.1: Raw Data File and Domain of Categorical Attributes

Categorical Attributes. In our exploration model, we distinguish categorical attributes from other textual or numeric attributes of the data file. Categorical attributes, characterized by their non-continuity and a distinct, limited set of values, play a pivotal role in many visualization techniques, such as bar charts or heat maps.

Let $\mathcal{A}_C \subseteq \mathcal{A}$ denote the *categorical attributes* of the objects. Each categorical attribute A_C is represented as a finite set of values $A_C = \{v_1, v_2, ...v_n\}$, which defines the domain of the attribute, i.e., $dom(A_C)$.

Example 1. [***Raw Data File & Objects***] Figure 3.1a presents a sample of a raw file containing five entries/objects (o_1 - o_5). Each entry *represents a signal measurement* and contains information regarding the: *geographic location* (Lat, Long), *signal strength* (Signal) and *network bandwidth* (Width), as well as network and device characteristics which take *categorical values* such as: device *brand*, *network provider*, and *network technology* (Net). To gain a better understanding of the categorical attributes, Figure 3.1b presents the domain of each categorical attribute in the data file.

Further, for each object o_i, there exists a *file pointer* f_i that indicates the offset of o_i from the beginning of the file. This file pointer can be utilized to access the attributes of o_i in a random-access manner.

3.3 Visual Exploration Model

Given a raw data file $\mathcal{F}$ containing a set of d-dimensional objects, the user arbitrarily selects[1] two attributes $A_x, A_y \in \mathcal{A}$, with numeric values that can be mapped to the X and Y axis of a 2D visualization layout (e.g., a map, scatter diagram). The A_x and A_y attributes are denoted as *axis attributes*, while the rest as *non-axis*. The

[1] We assume that the user is familiar with the schema of the data file; otherwise, as a first step, they may have a preview of it, in terms of loading a small sample.

non-axis attributes include all the other numeric or textual attributes of the data file, including the categorical attributes A_C.

The user selects to visualize a rectangular area $\Phi = (I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N_\Phi)$, called *visualized area*, which is defined by the two intervals $I_x = [x_1, x_2]$ and $I_y = [y_1, y_2]$ over the axis attributes A_x and A_y, respectively; i.e., Φ corresponds to the 2D area $I_x \times I_y$. The visualized area, contains a set of *visible objects* $\mathcal{O}_\Phi \subset \mathcal{O}$, for which the values of their axis attributes fall within the ranges of that area. Note that the mapping of the position (x, y) of the objects in the visualized area to their values A_x and A_y in the data is linear, e.g., spatial coordinates or any other affine mapping.

Each object $o_i \in \mathcal{O}_\Phi$ is associated with a set of visual annotations D_Φ presenting values from a set of $\{A_1, A_2, ...A_k\}$ non-axis attributes. Further, Φ can be linked to visual annotations G_Φ in order to differentiate objects based on their values for specific categorical attributes $\mathcal{C}$. Finally, Φ can be associated with a set of visual annotations N_Φ obtained by applying a collection of N aggregate functions to either all objects $\mathcal{O}_\Phi$ or groups of objects determined by G_Φ. Note that, the $\mathcal{O}_\Phi$, D_Φ, G_Φ and N_Φ can be empty sets.

We define a visual operation $VO : \Phi \rightarrow \Phi'$ as a 2D transformation on the visualized area, which transforms it to a new area $\Phi' = (I'_x, I'_y, \mathcal{O}'_\Phi, D'_\Phi, G'_\Phi, N'_\Phi)$. The following basic *visual operations/interactions* are considered:

- *render*: visualizes all objects contained in the visualized area. Formally: $VO_{render} : \Phi(I_x, I_y, \varnothing, D_\Phi, G_\Phi, N_\Phi) \rightarrow \Phi'(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N_\Phi)$. Note that the objects may be visualized as points or other visual elements.

- *move*: translates the boundary of the visualized area with shift constants k_x and k_y (i.e., number of pixels) on the X and Y axis, respectively. Formally: $VO_{move} : \Phi(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N_\Phi) \rightarrow \Phi'(I'_x, I'_y, \mathcal{O}'_\Phi, D'_\Phi, G'_\Phi, N'_\Phi)$, where $I'_x = [x_1 + k_x, x_2 + k_x]$, $I'_y = [y_1 + k_y, y_2 + k_y]$

- *zoom in/out*: zooms in/out the boundary of the visualized area keeping the point $\phi = (\phi_x, \phi_y)$ inside Φ as fixed point with a zoom factor $z\%$, with $z \in \mathbb{R}^+$. Formally: $VO_{zoom} : \Phi(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N_\Phi) \rightarrow \Phi'(I'_x, I'_y, \mathcal{O}'_\Phi, D'_\Phi, G'_\Phi, N'_\Phi)$, where $I'_x = (\phi_x - \sqrt{z}\frac{|I_x|}{2}, \phi_x + \sqrt{z}\frac{|I_x|}{2})$, $I'_y = (\phi_y - \sqrt{z}\frac{|I_x|}{2}, \phi_x + \sqrt{z}\frac{|I_x|}{2})$. VO_{zoom} corresponds to Zoom in operation when $0 < z < 1$, and to Zoom out when $z > 1$. Note that this operation assumes a scale on the X and Y coordinates and a subsequent translation to keep the area center ϕ fixed.

- *filter*: excludes objects visualized in Φ, based on conditions over the non-axis attributes. Formally: $VO_{filter} : \Phi(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G'_\Phi, N_\Phi) \rightarrow \Phi'(I_x, I_y, \mathcal{O}'_\Phi, D'_\Phi, N'_\Phi)$, where $\mathcal{O}'_\Phi \subset \mathcal{O}_\Phi$.

- *details*: visualizes annotations with values for non-axis attributes on every object included in Φ. Formally: $VO_{details} : \Phi(I_x, I_y, \mathcal{O}_\Phi, \varnothing, G_\Phi, N_\Phi) \rightarrow \Phi'(I_x, I_y, O_\Phi, D_S, N_\Phi)$.

- *group*: finds groups of objects based on one or more categorical attributes, i.e., similar to the group-by operation defined in SQL. The visual result of this operation could be some visual annotation distinguishing objects based on their group. Formally: $VO_{group} : \Phi(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, \varnothing, N_\Phi) \rightarrow \Phi'(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N'_\Phi)$, where the objects in $\mathcal{O}'_\Phi$ are partitioned into groups based on their categorical attributes. The resulting groups are represented visually within Φ'.

- *analyze*: computes aggregate values for all objects or groups of objects included in Φ and visualizes them appropriately as annotations of the entire area or the groups within it. For example, the aggregate values per group may be visualized through suitable techniques like bar charts, pie charts or heat maps, depending on the number of attributes involved in the *group* operation. Formally: $VO_{analyze} : \Phi(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, \varnothing) \rightarrow \Phi'(I_x, I_y, \mathcal{O}_\Phi, D_\Phi, G_\Phi, N'_\Phi)$, where N'_Φ includes aggregate values computed over either all objects or groups of objects in $\mathcal{O}'_\Phi$.

These operations may be combined in a sequence, e.g., zoom in a region and then filter the presented objects. Subsequent user actions form the *user's exploration model*, e.g., the user first renders a specific area Φ and then moves to render a neighboring area Φ'. Thus, a user's exploration model is a finite ordered set of visual operations applied by the user on the 2D space.

3.4 Exploratory Query

Considering the aforementioned visual operations, we proceed with mapping them to data-access operators, which operate on the underlying data file. Data-access operators are essentially the building blocks of a single query applied on the data, which we call *exploratory query*. In what follows, we formulate this notion and provide the definition of each operator. Next, we provide the mapping of visual to data access operators.

Given a set of objects $\mathcal{O}$ and the axis attributes A_x and A_y, an *exploratory query* Q over $\mathcal{O}$ is defined by the tuple $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{G}, \mathsf{N} \rangle$, where:

– *Selection clause* S: defines a 2D range query (i.e., window query) specified by two intervals I_x and I_y over the axis attributes A_x and A_y, respectively. The *Selection clause* is denoted as $\mathsf{S} = (I_x, I_y)$ and its intervals are $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. This clause selects the objects $\mathcal{O}_\mathsf{S} \subseteq \mathcal{O}$, for which the values of their axis attributes fall within the respective intervals, $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. The *Selection clause is mandatory* in a query Q, while the remaining clauses are *optional*.

– *Filter clause* F: defines a set of conjunction conditions that are applied *on the non-axis attributes*. The *Filter clause* is defined as $\mathsf{F} = \{F_1, F_2, ...F_k\}$, where a condition F_i is a predicate involving an atomic unary or binary operation over object attributes and constants. The Filter clause is applied over the selected objects $\mathcal{O}_\mathsf{S}$, returning the objects $\mathcal{O}_Q$ that satisfy the F conditions.

– *Details clause* D: defines a set of non-axis attributes $\mathsf{D} = \{A_1, A_2, ...A_k\}$, for which the values of the objects O_Q, will be returned by the query.

– *Group-by clause* G: defines a set of categorical attributes $\mathsf{G} = \{A_1, A_2, ...A_k\}$ with $A_i \in \mathcal{C}$, which are used in a group-by operation. Given a set of objects O and an attributes set $\mathcal{C}$, the *group-by operation* partitions O into a set of distinct groups, denoted as $\mathcal{G}_\mathcal{O}^\mathcal{C}$, based on the different combinations of the values of the $\mathcal{C}$ attributes in the O objects. Thus, here, the *Group-by clause* G performs a group-by operation based on its attributes, over the objects satisfying the filter $\mathcal{O}_Q$, resulting in the groups $\mathcal{G}_{\mathcal{O}_Q}^\mathsf{G}$.

– *Analysis clause* L: defines two sets of algebraic aggregate functions (e.g., count, mean) [38], where each of them is applied over a set of numeric attributes, returning a single numeric value. Particularly, the *Analysis clause* defines two sets of functions: (1) L_Q that are computed over the *objects* $\mathcal{O}_Q$ *returned by the query*; and (2) L_G that are computed over *each group of objects* resulted by the group-by operations. Thus, the analysis clause is defined as: $\mathsf{L} = (L_Q, L_\mathsf{G})$.

Note that, the support of algebraic aggregate functions in our model enables the computation of a large number of complex statistics, e.g., Pearson correlation, covariance.[2]

Intuitively, the Selection and Filter clauses apply restrictions (the equivalent of selection in relational algebra) to the entire space of objects, resulting in a set of qualifying objects $\mathcal{O}_Q$, which is visually presented. For each object in $\mathcal{O}_Q$, the values of the attributes included in the Details clause will be returned. Then, Group-by clause evaluates group-by operations over the $\mathcal{O}_Q$ objects. Finally, the set of aggregate functions of the Analysis clause is computed over the objects of $\mathcal{O}_Q$, and the objects' groups generated by the Group-by clause.

The *semantics of query execution* involves the evaluation of the different clauses of the query in the following order: (1) *Selection*; (2) *Filter*; (3) *Details*; (4) *Group-by*; (5) *Analysis*.

Mapping User Interactions to Exploratory Queries.

A user interaction can be mapped to clauses of an exploratory query. Table 3.1 presents the correspondences for the six aforementioned visual operations. Specifically, the *render*, *move*, and *zoom* operations are implemented by the *Selection clause*; the *render* operation sets the *Selection* intervals I_x and I_y equal to the region of the visualized area, *move* sets the intervals equal to the new intervals of the shifted area and *zoom in/out* operations set the *Selection* intervals to the new coordinates of the contained/containing visualized regions, respectively. Finally, the *filter*, *details*, *group*, and *analyze* operations are implemented by the query's *Filter*, *Details*, *Group-by* and *Analysis* clauses, respectively.

Query Result. The *result* $\mathcal{R}$ *of an exploratory query* Q over $\mathcal{O}$ is defined as $\mathcal{R} = (\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{L_Q}, \mathcal{V}_\mathsf{G})$, where:

(1) $\mathcal{V}_{x,y,\mathsf{D}}$ is a set of tuples corresponding to the objects $\mathcal{O}_Q$ returned by the query. For each object, its tuple contains: (a) the values of the axis attributes A_x and A_y; and (b) the values of the attributes D defined in the Details clause. Formally, $\mathcal{V}_{x,y,\mathsf{D}} = \{\langle o_i : \alpha_{i,x}, \alpha_{i,y},\ \ \alpha_{i,A_1}, ...\alpha_{i,A_k}\rangle, \forall o_i \in \mathcal{O}_Q\}$, where $\{A_1, ...A_k\} = \mathsf{D}$.

(2) $\mathcal{V}_{L_Q}$ is a list of the numeric values produced by the aggregate functions L_Q over the objects $\mathcal{O}_Q$. Formally, $\mathcal{V}_{L_Q} = \{\ell_1(\mathcal{O}_Q), \ell_2(\mathcal{O}_Q), ...\ell_k(\mathcal{O}_Q)\}$, $\forall \ell_i \in L_Q$.

(3) $\mathcal{V}_\mathsf{G}$ contains the results of the group-by clause. Particularly, $\mathcal{V}_\mathsf{G}$ is a set of tuples, where each tuple corresponds to a g_i group from $\mathcal{G}^\mathsf{G}_{\mathcal{O}_Q}$. Each tuple contains: ($a$) the values of the attributes G defined in the group-by clause; and (b) the results of the aggregate functions L_G (computed over g_i). Formally, $\mathcal{V}_\mathsf{G} = \{\langle g_i : a_{i,A_1}, ...a_{i,A_k},\ \ \ell_1(g_i), ...\ell_z(g_i)\rangle, \forall g_i \in \mathcal{G}^\mathsf{G}_{\mathcal{O}_Q}\}$, where $\{A_1, ...A_k\} = \mathsf{G}$ and $\{\ell_1, ...\ell_z\} = L_\mathsf{G}$.

Example 2. [***Exploratory Query***]

Figure 3.2 presents a 2D representation of 12 objects from the file presented in Figure 3.1a, in which the attributes *Lat* and *Lon* have been selected as the *axis*

[2] More than 90% and 75% of the statistics supported by SciPy and Wolfram, respectively, are defined as algebraic aggregate functions [94].

Table 3.1: Correspondences between Visual Operations and Exploratory Queries *

Description	Visual Operation	Exploratory Query
Render the objects included in the visualized 2D area Φ defined by the intervals I_x, I_y.	**render** Φ	$\mathsf{S} = (I_x, I_y)$
Move the visualized area Φ to a new Φ′.	**move** from Φ to Φ′ $\Phi' = I'_x \times I'_y$	$\mathsf{S} = (I'_x, I'_y)$
Zoom in/out over the visualized area Φ, having as zoom center the point ϕ inside Φ, and a zoom factor z%. Zoom in: $0 < z < 1$ Zoom out: $z > 1$	**zoom in/out** z% over Φ with center ϕ $\phi = (\phi_x, \phi_y), \quad z \in \mathbb{R}^+$	$\mathsf{S} = (I'_x, I'_y)$ $\mathsf{S}.I'_x = [\phi_x - \sqrt{z}\frac{\lvert I_x \rvert}{2},\ \phi_x + \sqrt{z}\frac{\lvert I_x \rvert}{2}]$ $\mathsf{S}.I'_y = [\phi_y - \sqrt{z}\frac{\lvert I_y \rvert}{2},\ \phi_y + \sqrt{z}\frac{\lvert I_y \rvert}{2}]$
Filter the objects included in the visualized area Φ, by applying the set of conditions $\{c_1, c_2, ...c_k\}$	**filter** the objects inside Φ, $\{c_1, c_2, ...c_k\}$	$\mathsf{S} = (I_x, I_y)$ $\mathsf{F} = \{c_1, c_2, ...c_k\}$
Presents the values of the attributes $\{A_1, A_2, ...A_k\}$ for the objects included in the visualized area Φ.	**detail** the objects inside Φ, $\{A_1, A_2, ...A_k\}$	$\mathsf{S} = (I_x, I_y)$ $\mathsf{D} = \{A_1, A_2, ...A_k\}$
Group the objects in the visualized area Φ based on one or more categorical attributes.	**group** the objects inside Φ, $\{A_1, A_2, ...A_k\}$ with $A_i \in \mathcal{C}$	$\mathsf{S} = (I_x, I_y)$ $\mathsf{G} = \mathcal{G}^{\mathcal{C}}_{\mathcal{O}_\Phi}$
Analyze the objects in the visualized area Φ, based on a set of functions $\{F_1, F_2, ...F_k\}$.	**analyze** the objects inside Φ, $\{F_1, F_2, ...F_k\}$	$\mathsf{S} = (I_x, I_y)$ $\mathsf{N} = \{F_1, F_2, ...F_k\}$

* Φ is the visualized 2D area $I_x \times I_y$

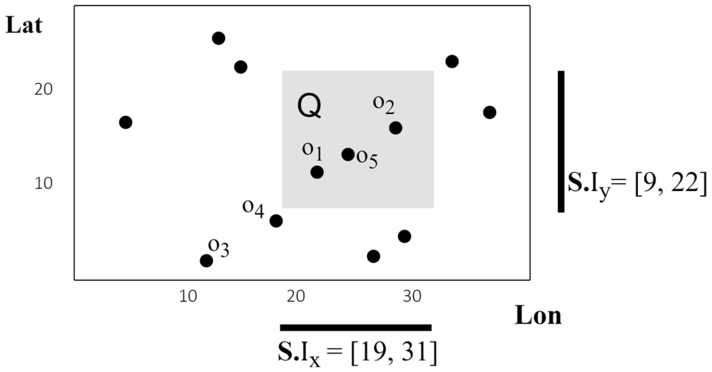


Figure 3.2: Exploratory Query

attributes A_X and A_Y, respectively.

Also, an exploratory query Q is presented in the figure. The *Selection clause* of Q is defined by the two intervals $\mathsf{S}.I_x$=[19, 31] and $\mathsf{S}.I_y$=[9, 22]. The query selects all objects contained in this 2D area. The objects O_S selected by the *Selection clause* are o_1, o_2, o_5. Assuming that the query has only a *Selection clause*, the result fetches only axis attribute values, i.e., $\mathcal{R} = (\mathcal{V}_{x,y,\mathsf{D}} = \langle o_1 : 21, 11\rangle, \langle o_2 : 29, 18\rangle, \langle o_5 : 23, 12\rangle)$.

If we enrich the query with a *Filter clause* $\mathsf{F} = Width > 5$, which applies a condition over the bandwidth attribute, i.e., $\mathsf{F}_A = Signal$, then the result will be $\mathcal{R} = (\mathcal{V}x, y, \mathsf{D} = \langle o_1 : 21, 11\rangle, \langle o_5 : 23, 12\rangle)$, as the o_2 is omitted due to its bandwidth value of 4. Furthermore, adding to the above query a *Details clause* $\mathsf{D} = Signal$, the result becomes $\mathcal{R} = (\mathcal{V}_{x,y,\mathsf{D}} = \langle o_1 : 21, 11, 3\rangle, \langle o_5 : 23, 12, 4\rangle)$.

Further, assume that the user wishes to group the objects based on their values for the categorical attribute *Signal*. After establishing the grouping, the user then defines an *Analysis clause* $\mathsf{L} = (L_Q, L_\mathsf{G})$ for the query. Here, the L_Q function set applies to the overall query, whereas the L_G function set applies to each group defined in the Group-by clause. In this example, for the overall query analysis clause L_Q, the user has specified a function $corr(Signal, Width)$ which calculates the *correlation* (i.e., Pearson correlation coefficient) between *Signal* and *Width*. This function is computed only over the objects included in the query result; i.e., o_1 and o_5. For each group, the user also specifies an analysis function, Avg($Signal$), in the group analysis clause L_G.

As a result, the query result now includes both overall and group-wise aggregate results, and becomes:

$\mathcal{R} = (\mathcal{V}_{x,y,\mathsf{D}} = \langle o_1 : 21, 11, 3\rangle, \langle o_5 : 23, 12, 4\rangle, \mathcal{V}_{L_Q} = \langle corr(Signal, Width) : 0.996\rangle,$ $\mathcal{V}_\mathsf{G} = \langle Avg(Signal) : \langle Veriz : 3, 3\rangle, \langle AT\&T : 4, 4\rangle\rangle$

3.5 Summary

This chapter introduced a formal model for the in-situ visual exploratory analysis that this thesis attempts to address. Essential concepts such as the structure of raw data files, attributes, and the stages involved in the targeted visual analysis were detailed. The model offers an in-depth examination of the connection between user interactions, visual and analytic operations, and query evaluation. The chapter also scrutinized the critical role of exploratory queries in defining various data analysis operations. With an established link between visual operations and these queries, the chapter provides a clear understanding of how user interactions connect with underlying data access and processing operations. This understanding serves as the foundation for presenting the novel techniques and index structures proposed in subsequent sections in the context of in situ visual exploration of large raw data files.

Chapter 4

Indexing for Efficient 2D Visual Exploration

4.1 Introduction

In this chapter, our primary objective is to enable efficient 2D user exploration scenarios over raw data files, utilizing 2D visualizations such as maps or scatter plots for user interaction. Considering the visual exploration model outlined in Chapter 3, we concentrate on addressing the challenges of efficiently executing 2D operations such as *render*, *move*, *zoom in/out*, as well as improving the efficiency of *analyze* operations to calculate aggregate statistics over the 2D window visualized by the user. To this end, we introduce a main-memory index designed explicitly for 2D *in-situ* visual exploration of large raw data. This index, named VALINOR (Visual AnaLysis Index On Raw data), features a hierarchical, tile-based structure that groups objects based on two numeric attributes (e.g., latitude and longitude for geospatial data). Augmented with aggregated metadata, the index offers enhanced analytic capabilities.

Following the introduction of the index, we describe a user-driven initialization algorithm that employs the user's first query and a locality-based probabilistic approach. This approach significantly accelerates the initial stages of user interaction by decreasing response time. We also propose a query-based adaptation technique that improves overall performance, specifically for analytic tasks. This technique involves incrementally refining the index structure based on user interaction, optimizing the utilization of I/O operations for updating index metadata. Further, to address potential memory limitations, we incorporate an eviction mechanism, allowing parts of the index to be stored on disk when necessary. Finally, to substantiate the theoretical propositions detailed in this chapter, we undertake an exhaustive experimental evaluation using both real-world and synthetic datasets and demonstrate that our methodology consistently surpasses comparative systems, often resulting in a 5-10 times speed increase.

4.2 VALINOR Design

4.2.1 Design Principles

The VALINOR index is a lightweight *tile-based multilevel* index, which is stored in memory and organizes the data objects of a raw file, into *tiles.* The index is constructed on-the-fly given the first user query and incrementally adjusts its structure to the user visual interactions. Each tile is constructed, during initialization, over specific ranges for the A_x and A_y axis attributes, by dividing the Euclidean space into initial tiles (see Sect 4.4.1 for the initialization method). Further, considering the distributivity of the employed aggregate functions, each tile contains metadata that allows efficient query evaluation. Subsequent user operations split these tiles into more fine-grained ones, thus forming a hierarchy of tiles. Overall, the design of our index relies on the following basic principles: (1) fast on-the-fly construction; and (2) effective metadata computations and storing, which in turn, offers efficient computation of aggregate functions. These principles are further enhanced by exploiting advanced methods in the context of user exploration scenarios.

4.2.2 Core Elements of the VALINOR Index

Object Entry. For an object o_i its *object entry* e_i is defined as $\langle a_{i,x}, a_{i,y}, f_i \rangle$, where $a_{i,x}, a_{i,y}$ are the values of the axis attributes and f_i the offset (a hex value) of o_i in the raw file.

Tile. A *tile* t is a part of the Euclidean space defined by two left-closed, right-open intervals $t.I_x$ and $t.I_y$. In this work, we assume hierarchies of tiles (i.e., forest), although a hierarchy with a single root tile can also be defined. A tile can have an arbitrary number of *child nodes*, whereas *leaf tiles* are the tiles without child nodes. A non-leaf tile covers an area that encloses the area represented by any of its children: given a tile t with $t.I_x = [x_1, x_2)$ and $t.I_x = [y_1, y_2)$, for each child node t' of t, with $t'.I_x = [x'_1, x'_2)$ and $t'.I_x = [y'_1, y'_2)$, it holds that $x_1 \le x'_1, x_2 \ge x'_2, y_1 \le y'_1$ and $y_2 \ge y'_2$. In each level of the hierarchy, there are no overlaps between the tiles of the same level (i.e., disjoint tiles). Further, leaf tiles can appear at different levels in the hierarchy.

Each tile t is associated with a *set of object entries* $t.\mathcal{E}$, if it is a leaf tile, or a set of *child tiles* $t.C$, if it is a non-leaf tile. The set $t.\mathcal{E}$ is the set of object entities, such that for each $e_i \in t.\mathcal{E}$ its attribute values $a_{i,x}$ and $a_{i,y}$ fall within the intervals of the tile t, $t.I_x$ and $t.I_y$ respectively.

Synopsis metadata Apart from object entries, each tile t is associated with a set of *synopsis metadata* $t.\mathcal{M}$ which are aggregated or computed values computed from the $t.\mathcal{E}$ objects contained in the tile over their attributes. For simplicity, synopsis metadata is also referred to as *metadata.* As $t.\mathcal{M}_A$ we denote the set of attributes for which metadata has been computed for the tile t.

The synopsis metadata $t.\mathcal{M}$ of a tile t are numeric values calculated by algebraic aggregate functions, over all objects $t.\mathcal{E}$ in t. Exploring the synopsis metadata for a set of tiles $\mathcal{T}_k$, we can compute values for more complex algebraic aggregate functions, for the objects included in tiles $\mathcal{T}_k$. The main idea is that metadata are

Table 4.1: VALINOR Notation

Symbol	Description
$\mathcal{F}$	Raw data file
$\mathcal{O}$, o_i	Set of d-dimensional objects, an object
f_i	Position of o_i in the file $\mathcal{F}$
$\mathcal{A}$	List of attributes
A_j, $a_{i,j}$	the j$^{\text{th}}$ attribute of the list, the value of attribute A_j of the object o_i
A_x, A_y	Axis attributes
Φ, ϕ	2D visualized area, center of the visualized area
Q	Exploratory Query
S, F, D, N	Select, Filter, Details & Analysis clause
$\mathcal{O}_{\mathsf{S}}$, $\mathcal{O}_{\mathsf{Q}}$	Objects selected from S, Objects resulted from $\mathcal{O}_{\mathsf{S}}$ after evaluating F
$\mathcal{V}_{x,y,\mathsf{D}}$	Values of axis attributes along with Details attributes' values
$\mathcal{V}_{\mathsf{N}}$	Numeric values resulted from the Analysis clause
$(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$	Query result
$\mathcal{I}$	VALINOR index
$\mathcal{T}$, t	Set of tiles in the index, a tile
$t.I_x$, $t.I_y$	Intervals of tile t
$t.\mathcal{E}$	Object entries in tile t
$t.\mathcal{M}$	Metadata of tile t
$\mathsf{IP}, \mathsf{AP}, \mathsf{MH}$	Initialization, Adaptation policy & Metadata handler
$t.\mathcal{E}_{\mathsf{S}}$	Objects of t that are included in the 2D area specified by S
R_t^{S}	2D area of t that overlaps with the area specified by S
t_Q	Query subtile

defined at the level of a single tile (i.e., for the objects of a tile, we carry the aggregate values of several aggregate functions over all the objects of a tile). When the tile has children, we can compute the aggregate statistics for the tile, from the aggregate statistics of its children. Naturally, this requires the restriction of the employed aggregate functions to *algebraic* ones, which by definition can distribute the computation of the aggregate statistic over a set to a composition of aggregate statistics over its subsets [59]. Specifically, we employ functions like *count*, *sum*, *mean*, $sumOfSquaresOfDeltas$, *min*, *max* over the objects of a tile. Whenever an aggregate computation is required over tiles that are fully contained in the query, their existing stats can be exploited directly, without having to go to the disk to retrieve the necessary columns and compute them.

4.2.3 VALINOR Index Definition

Given a raw data file $\mathcal{F}$ and two axis attributes A_x, A_y, the index organizes the objects into hierarchies of non-overlapping rectangle tiles based on its A_x, A_y values. Specifically, the VALINOR *index* $\mathcal{I}$ is defined by a tuple $\langle \mathcal{T}, \mathsf{IP}, \mathsf{AP}, \mathsf{MH} \rangle$, where $\mathcal{T}$ is the set of *tiles* defined in the index; IP is the *initialization policy* defining the methods to compute the sizes of tiles and construct the tiles during the initialization phase;

AP is the *adaptation policy* defining the method for reconstructing the index

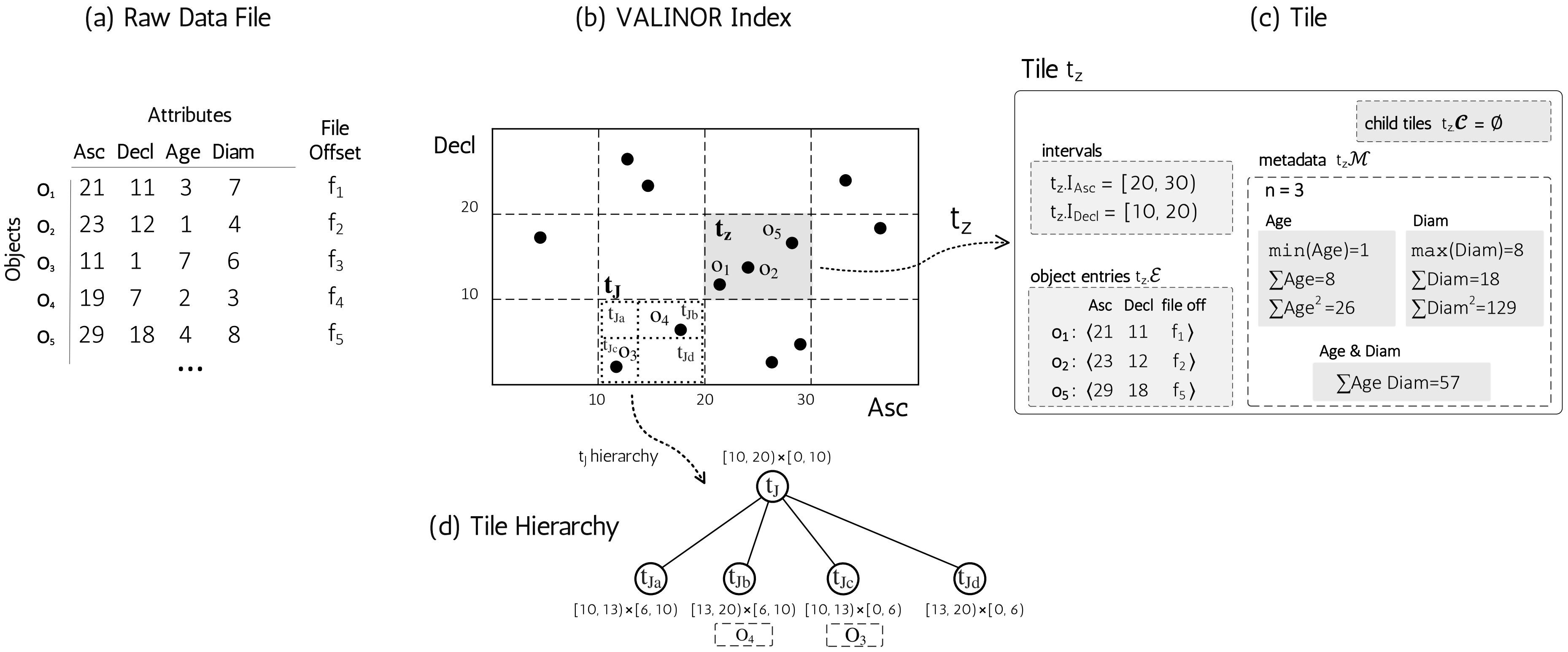


Figure 4.1: The VALINOR Index Overview

and reorganizing object entries following user's interaction; and MH is the *metadata handler* which performs the computations in the metadata stored in each tile.

Example 3. [***VALINOR Index***] Figure 4.1(a) presents a sample of a raw data file, containing five objects (o_1-o_5), where each object represents an observation of a sky object, such as a star. Each object is described by four *attributes.* The attributes *Asc* and *Decl* correspond to right ascension and declination, respectively, measured in degrees. Practically, right ascension corresponds to terrestrial longitude and declination to geographic latitude; their combination gives the position of an object in the sky. The *Age* attribute measures the age of the star in billion years, and the diameter (*Diam*) measures the diameter in km.

Assume that the attributes *Asc* and *Decl* have been selected as the *axis attributes* A_X and A_Y, respectively. Figure 3(b) presents a version of the VALINOR index, which (in the upper-level) divides the 2D space into 4×3 equally sized disjoint tiles, and the tile t_j is further divided into 2×2 subtitles of arbitrary sizes. The multilevel structure of the tile t_j is presented as a hierarchy in the Figure 4.1(d). Figure 4.1(c) presents the contents of a tile t_z, highlighted with grey color in the index. For each tile, the index stores its intervals $t_z.I_{Asc}$ and $t_z.I_{Decl}$, the object entries $t_z.\mathcal{E}$ contained in this tile and a set of metadata computed over axis or non-axis attributes of the contained objects. In the example, t_z contains o_1, o_2 and o_5.

Furthermore, for each object in the tile, the index stores the values of the axis attributes along with the offset pointing to the position of the object in the file. For example, the entry for the object o_1 is $\langle 31, 11, f_1 \rangle$, where 33 and 11, are the *Asc* and *Decl* values of the o_1, respectively.

Finally, in our example the index stores for t_z the number of enclosed objects ($n = 3$), as well various statistics for the two non-axis attribute *Age* and *Diam*, such as the *min*, *max*, *sum* values, the *sum of squares* and the *sum of their product.*

4.2.4 Tiles-Query Spatial Relations

Consider an exploratory query Q (Sec. 3.4), with S being its Selection clause. Recall that the Selection clause defines a 2D range query (i.e., window query) over the axis attributes A_x and A_y, respectively. Also, let $\mathcal{T}$ be the tiles defined in the VALINOR index.

Considering the spatial relations between the Selection clause of the query and tiles included in the VALINOR, we denote as $\mathcal{T}_\mathsf{S} \in \mathcal{T}$ the *leaf tiles that overlap with the 2D area* (plane) specified by S. Also, the tiles $\mathcal{T}_\mathsf{S}$ are divided into two disjointed tile sets $\mathcal{T}_{\mathsf{S}_f}$ and $\mathcal{T}_{\mathsf{S}_p}$, which denote the tiles of $\mathcal{T}_\mathsf{S}$ that are *fully-* and *partially-contained* in S, respectively.

Further, given a tile $t \in \mathcal{T}_\mathsf{S}$, we denote the *object entities* of t that are *included in the 2D area* specified by S as $t.\mathcal{E}_\mathsf{S}$. Note that, in case that t is a fully-contained tile, then $t.\mathcal{E}_\mathsf{S} = t.\mathcal{E}$.

Additionally, given a tile $t \in \mathcal{T}_\mathsf{S}$, we denote the *plane of t that overlap with* S as R_t^S. Hence, in case that a tile t is fully-contained by the query, then R_t^S corresponds to the area defined by t.

4.2.5 Implementation Details and Practical Considerations

To make our implementation work, we have adopted several design choices and assumptions. Firstly, we presume that the data in a CSV file are arranged in homogeneous records delineated by a new line character, excluding any headers in the file; these records share an identical schema, hence contain an equal number of attributes. Each record's attributes are divided by a comma, succeeded by a new line symbol indicating the beginning of the next record in the file.

In our system, the offset of the record corresponds to its first character's position in the file, defined as the hex value of the location immediately following the new line delimiter. To maintain a connection between each tile and its corresponding records in the data file, we retain a list of each record's offset (hex value) start.

During the raw file parsing phase, we optimize tokenizing and parsing costs by only processing the attributes necessary for a query. We stop tokenizing as soon as the last required attribute for the query or initialization is found in the row.

From the user's side, the necessary input is minimal. Users only need to specify the delimiter of the CSV file (e.g., comma or tab), identify the axis attributes, and provide a reasonable estimate of their ranges. This approach eliminates the need for scanning the raw file to discern these ranges.

In the context of the system's structure, each object within a tile consists precisely of two float values (x, y coordinates) and a long value (offset). Importantly, the index tiles are not of uniform size.

4.2.6 Grid or R-Tree?

The insightful reader might wonder what are the benefits of following a grid-based approach, rather than an R-Tree one. We surveyed the literature on the comparison of grid files and R-trees. As already mentioned, regarding the construction of an R-tree, its inherent objectives (i.e., tree balance, space coverage, node overlaps, fill guarantees) result in the need for substantial memory and time resources (even main memory implementations), which makes them inappropriate for enabling the users to quickly start exploring and interacting with the data, as in the case of *in-situ* data exploration. Hence, one major limitation of using spatial structures in our scenario is related to efficient construction phase.

The expensive construction phase of several (main-memory) spatial indexes is also validated by several studies. Regarding our case, considering that the construction cost of the initial VALINOR version, is similar to the construction cost of a grid structure [72]. In this context, the better performance of main-memory grid over several spatial structures (e.g., R-tree variances, quadtree) is demonstrated in several recent experimental studies. In more details, several studies have demonstrated that main memory grid indexes have considerable better performance on construction phase [86, 98, 88]. Further, some studies suggest that grid indexes have better performance even over the R-tree versions that use efficient bulk loading methods [98, 88]; i.e., STR [60] and Hilbert R-Tree [55].

Regarding the query performance, recent studies [85, 53, 86], show that the grid index, have noticeably better performance in range and kNN queries, as well as update operations, compared to R-tree variances and quad-tree, *when the indexing is performed in 2 dimensions and the indexes are stored in main memory.* Additionally,

Algorithm 1. VALINOR Initialization & First Query Evaluation ($\mathcal{F}$, A_x, A_y, Q_0)

```
Input: $\mathcal{F}$: raw data file;  $A_x$, $A_y$: X and Y axis attributes;  $Q_0$ $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$: first query
Parameters: IP: initialization policy; MH: metadata handler
Output: $\mathcal{I}$: initialized index;  $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$: first query result $\mathcal{R}$
Variables: $\boldsymbol{V}$: the attribute values used in Analysis clause computation

1  $\boldsymbol{V} \leftarrow \varnothing$
2  $\ell_{x_0}, \ell_{y_0} \leftarrow$ IP.computeInitialTileSize($A_x, A_y$)    //determine the initial tile size
3  $\mathcal{I}, \mathcal{T} \leftarrow$ IP.constructInitialTiles($\ell_{x_0}, \ell_{y_0}$)    //determine the intervals of the tiles and construct the tiles $\mathcal{T}$ that initialize the index $\mathcal{I}$
4  foreach $o_i \in \mathcal{F}$ do    //read objects from file, assign them to the constructed tiles, and evaluate the first query $Q_0$
5      read $a_{i,x}$, $a_{i,x}$ from $\mathcal{F}$
6      $f_i \leftarrow$ offset of $a_{i,1}$ in $\mathcal{F}$
7      append $\langle a_{i,x}, a_{i,y}, f_i \rangle$ to tile entries $t.\mathcal{E}$, where $t \in \mathcal{T}$ determined from $a_{i,x}, a_{i,y}$ and $t$ intervals    //assign the object $o_i$ to tiles $t$
8      MH.updateMetadata($t.\mathcal{M}, o_i$)
9      if $o_i$ included in Selection clause S and satisfies the Filter clause F then    //evaluate the query
10         $\alpha_{i,A_{\mathsf{D}_1}}, ...\alpha_{i,A_{\mathsf{D}_k}} \leftarrow$ for $o_i$ read the values of the attributes $\mathsf{D}_1, ...\mathsf{D}_k$ referred in the Details clause D
11         insert $\langle o_i : \alpha_{i,x}, \alpha_{i,y}, \alpha_{i,A_{\mathsf{D}_1}}, ...\alpha_{i,A_{\mathsf{D}_k}} \rangle$ into $\mathcal{V}_{x,y,\mathsf{D}}$    //insert a result tuple into results
12         insert into $\boldsymbol{V}$ the values of $o_i$ for the attributes $\mathsf{N}_A$ referred in the Analysis clause A
13 $\mathcal{V}_{\mathsf{N}} \leftarrow$ use the values of $\boldsymbol{V}$ to compute the statistics of the Analysis clause A
14 return $\mathcal{I}$,  $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$
```

[86] concludes that grid index is surprisingly robust to varying parameters of the query workloads.

4.3 Query Processing over the VALINOR Index

This section describes the process for the evaluation of exploratory queries over the index. It first presents the initialization of the index, which is constructed by the first query posed, and then it describes the evaluation of subsequent queries performed over the initialized index.

4.3.1 Index Initialization & First Query Evaluation

In our approach, we do not consider any loading phase for the index construction, but rather the index is constructed on-the-fly the first time the user requests to visualize a part of the file. Considering an interactive scenario, the index construction should entail a small overhead in the response time of the first query. Thus, a lightweight version of the index is constructed, which corresponds to a flat tile structure, by parsing the raw file once.

Algorithm 1 describes the initialization phase. The algorithm takes as input, the raw file $\mathcal{F}$, the axis attributes A_x, A_y, and the first exploratory query Q_0, and provides as output, the initialized index $\mathcal{I}$ and the results of the first query $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$.

First, the initialization policy IP uses the computeInitialTileSize method to determine an initial tile size ℓ_{x_0}, ℓ_{y_0} (*line* 2). Then, using this initial size, the constructIni-

Algorithm 2. VALINOR Query Processing ($\mathcal{I}$, Q, $\mathcal{F}$)

```
Input: $\mathcal{I}$: index (initialized);  $Q\ \langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$: query;  $\mathcal{F}$: raw data file
Variables: $\mathcal{O}_{\mathsf{S}}$: objects selected from Select clause; $\mathcal{T}_{\mathsf{S}}$: leaf tiles that overlapped with the Select
           clause;
           $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$: leaf tiles for which file access is required; $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$: tiles resulted from $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ after splitting;
           $\mathcal{V}_{\mathsf{F}_A}$: values of the attributes included in the Filter clause; $\mathcal{V}_{\mathsf{D}}$: values of the attributes
           defined in the Details clause;
           $\mathcal{V}_{\mathsf{N}_A}$: values of the attributes required for the Analysis clause computation;
           $\mathcal{V}_{x,y,\mathsf{D}}$: objects of the result along with the detail values; $\mathcal{V}_{\mathsf{N}}$: numeric values resulted from
           the Analysis clause
Parameters: AP: adaptation policy; MH: metadata handler
Output: $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$: query result $\mathcal{R}$

$\mathcal{O}_{\mathsf{S}}$, $\mathcal{T}_{\mathsf{S}}$ ← evaluateSelectionClause ($\mathcal{I}$,S)
$\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ ← getTilesRequireFileAccess ($\mathcal{T}_{\mathsf{S}}$, $Q$)
$\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$ ← AP.adaptTiles ($\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, $\mathcal{O}_{\mathsf{S}}$)
if $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}} \neq \varnothing$ then
    $\mathcal{V}_{\mathsf{F}_A}$, $\mathcal{V}_{\mathsf{D}}$, $\mathcal{V}_{\mathsf{N}_A}$, ← readFile ($\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$, $\mathcal{O}_{\mathsf{S}}$, $Q$, $\mathcal{F}$)
if $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}} \neq \mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ then
    MH.updateMetadata ($\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$, $Q$, $\mathcal{V}_{\mathcal{A}_{\mathsf{F}}}$, $\mathcal{V}_{\mathsf{N}_A}$ )
$\mathcal{O}_Q$ ← evaluateFilterClause ($\mathcal{O}_{\mathsf{S}}$, $\mathcal{V}_{\mathsf{F}_A}$)
$\mathcal{V}_{x,y,\mathsf{D}}$ ← construct the tuples by combining $\mathcal{O}_Q$ and $\mathcal{V}_{\mathsf{D}}$
$\mathcal{V}_{\mathsf{N}}$ ← evaluateAnalysisClause ($\mathcal{O}_Q$, N, $\mathcal{V}_{\mathsf{N}_A}$)
return $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$
```

tialTiles method constructs the tiles $\mathcal{T}$ of the index, which corresponds to the initial flat structure of the index without any computed metadata on each tile.

For instance, an initial tile size can be either (1) given explicitly by the user (e.g., in a map the user defines a default scale of coordinates for the initial visualization); (2) provided by the visualization setting considering certain characteristics (e.g., screen size/resolution, visualization type) [52, 12, 89, 18]; or, (3) computed from the data in the raw file based on a binning technique that divides the data space into equal size tiles. We consider the latter as the baseline method for the initialization of the index. In Section 4.4.1, we propose an advanced method that determines and constructs varying tile sizes by considering the user exploration entry point, i.e., the position of the first user query in the 2D space.

In the next step, the algorithm scans once the file $\mathcal{F}$ (*loop in line* 4). For each object, the algorithm reads the attribute values of $a_{i,x}$, $a_{i,y}$ and the file offset f_i (*lines* 5 & 6). Then, it appends the object to the entries $t.\mathcal{E}$ of the corresponding tile t (*line* 7). The updateMetadata method considers the values of o_i to compute and update the metadata $t.\mathcal{E}$ of the tile t (*line* 8).

Next, the algorithm evaluates the query (*lines* 9-13). It first checks if the object o_i is included in the query result (*line* 9), i.e., whether o_i is selected by the Select clause, and satisfies the conditions of the Filter clause. Then, it reads the attribute values in the Details clause, constructs the result tuple of o (*line* 10), and inserts the tuple to the result set $\mathcal{V}_{x,y,\mathsf{D}}$ (*line* 11).

As a final step, the algorithm reads the attribute values of o_i (line 12) and computes the Analysis clause for each tile (*line* 13). Finally, the result of the first query and the initialized index are returned (*line* 14).

4.3.2 Query Processing Overview

The following process describes the evaluation of all subsequent queries. An overview of the query evaluation is presented in the Algorithm 4 and details for each operator are provided in following subsections. Algorithm 4 takes as input, the initialized index, an exploratory query and the raw file. The algorithm returns (a) the values of the two axis attributes of the objects in the result set along with the values of the attributes defined in the *Detail* clause of the query, and, (b) the values computed for each tile in the Analysis clause.

First, the Selection clause is evaluated (*line* 1), using the evaluateSelectionClause procedure (Proc. 1). Given a query Q, this procedure first looks up the index $\mathcal{I}$ and determines the leaf tiles $\mathcal{T}_{\mathsf{S}}$ overlapping with the Selection clause of the query. For each tile, we examine its objects and select the objects $\mathcal{O}_{\mathsf{S}}$, contained in the query window. The getTilesRequireFileAccess procedure (Proc. 2) determines the leaf tiles $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}} \in \mathcal{T}_{\mathsf{S}}$ for which access to the raw file is required (*line* 2). In the next step (*line* 3), each leaf tile $t \in \mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ is examined for splitting, based on the adaptation procedure adaptTiles (Proc. 3). The splitting process results in a new set of tiles $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$, which is a super-set of $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, containing also the subtiles created by the splitting (as well as the tiles' hierarchies info).

Next, the procedure readFile (Proc. 4) retrieves from the file the objects $t_{\mathcal{E}_{\mathsf{S}}}$ of each leaf tile t from $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$; specifically it retrieves the values of all attributes $\mathcal{V}_{\mathsf{D}}$, $\mathcal{V}_{\mathsf{F}_A}$, $\mathcal{V}_{\mathsf{N}_A}$ required for the evaluation of the Details, Filter, and Analysis clauses, respectively (*line* 5).

If tile splitting is performed (*line* 6), the updateMetadata procedure computes and updates the metadata in tiles $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$ (*line* 7). Finally, the Filter (*line* 8), Details (*line* 9) and Analysis (*line* 10) clauses are evaluated.

Example 4. [***VALINOR Query Processing***] In this example we assume an exploratory query Q with the following clauses: (1) *Selection clause*: $\mathsf{S}.I_x$=[19°, 31°], $\mathsf{S}.I_y$=[9°, 22°]; (2) *Filter clause*: $\mathsf{F} = \{Diam < 5 \text{ km}\}$; and (3) *Analysis clause*: $\mathsf{N} = \{corr(Age, Diam), Avg(Age)\}$. Further, we assume the index described in Example 3 and presented in Figure 4.1.

The query processing procedure is depicted in Figure 4.2. We assume that the index is already initialized (i.e., the Q is not the first query). ❶ depicts the index before the query Q is posed, whereas ❷ depicts the updated index after Q evaluation.

First, we have to evaluate the Selection clause. We identify the tiles that overlapped with the query; i.e., t_1, t_2, t_3, t_4. Then, for each of these tiles, we select these objects that are selected by the query; i.e., o_1, o_2, o_4.

Next, we have to identify for which of the overlapped tiles we have to access the file. In our case, the tiles t_1 and t_4 are omitted from the process that follows, since these tiles do not include any of the selected objects. Both tiles t_2 and t_3 are partially contained in the query. As a result, we do not have the metadata for the selected objects to compute the Analysis and Filter clause defined in the query. Recall that, the metadata is computed and stored per tile.

Hence, we have to access the file for objects o_1, o_2, o_4, and read the attribute values required for the evaluation and, particularly, the attributes Diam and Age that are used in the Filter and/or Analysis clause. Using the retrieved values, we can evaluate all the parts of the query.

Along with the query evaluation, the index structure is adapted via splitting.

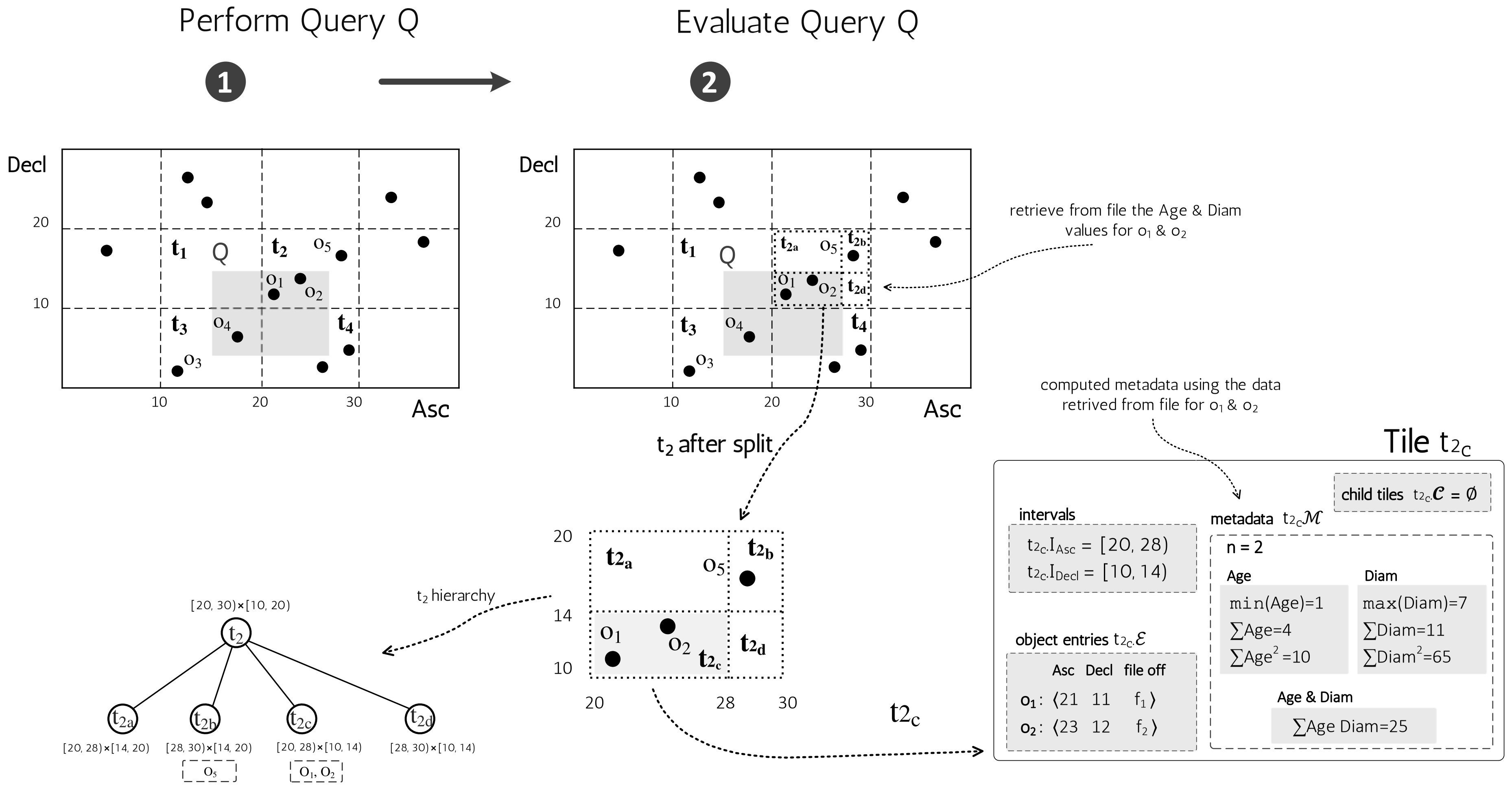


Figure 4.2: Query Processing over VALINOR Index

```
Procedure 1: evaluateSelectionClause($\mathcal{I}$, S)
    Input: $\mathcal{I}$: index;    S : Selection clause of the query
    Output: $\mathcal{O}_\mathsf{S}$: objects selected from Selection clause;    $\mathcal{T}_\mathsf{S}$: leaf tiles that overlapped with Selection
            clause
1   $\mathcal{T}_\mathsf{S} \leftarrow$ getSelectOverlappedLeafTiles $(\mathcal{I}, \mathsf{S})$
2   forall $t \in \mathcal{T}_\mathsf{S}$ do
3       $t.\mathcal{E}_\mathsf{S} \leftarrow$ getSelectedObjectsFromTile $(t, \mathsf{S})$
4       insert $t.\mathcal{E}_\mathsf{S}$ into $\mathcal{O}_\mathsf{S}$
5   return $\mathcal{O}_\mathsf{S}$, $\mathcal{T}_\mathsf{S}$
```

In our example, the tile t_2 is split into four disjoint subtiles t_{2_a}, t_{2_b}, t_{2_c}, t_{2_d}. As previously mentioned, we have to access the file for the objects o_1 and o_2, which are the objects included in subtile t_{2_c}. Using the retrieved attribute values, we can compute the metadata for the subtile t_{2_c}. Overall, during the query processing, we evaluate the query; and we construct subtiles and compute metadata for the constructed subtiles. A detailed example for the splitting process is presented later in the adaptation section (Sect. 4.4.2, Ex. 5).

4.3.3 Selection Clause Evaluation

In order to evaluate the Selection Clause over the index (Alg. 2, *line* 1), we have to identify the $\mathcal{O}_\mathsf{S}$ objects by accessing the leaf tiles $\mathcal{T}_\mathsf{S}$ which overlap with the window query specified in the Selection Clause of Q.

First, we define the following simple function used in the Selection Clause evaluation.

- getSelectOverlappedLeafTiles$(\mathcal{I}, \mathsf{S})$: This function returns the leaf tiles $\mathcal{T}_\mathsf{S}$ which overlap with the Selection clause S of the query. It identifies the highest-level overlapped tiles. Then, for each tile, it traverses the hierarchy to determine the overlapped leaf tiles $\mathcal{T}_\mathsf{S}$.

- getSelectedObjectsFromTile(t, S): This function scans all objects $t.\mathcal{E}$ of a tile t and returns the objects $t.\mathcal{E}_\mathsf{S}$ that are included in the Selection clause S of the query.

  In case that the tile t is fully-contained in the Selection clause, the returned objects $t.\mathcal{E}_\mathsf{S}$ correspond to all objects included in the t; i.e., $t.\mathcal{E}$. On the other hand, if t is partially-contained, the returned objects $t.\mathcal{E}_\mathsf{S}$ are the objects included in the overlapped 2D area R_t^S.

The evaluation of Selection clause is described in the evaluateSelectionClause procedure (Proc. 1). First, it identifies the leaf tiles $\mathcal{T}_\mathsf{S}$ using the function getSelectOverlappedLeafTiles (*line* 1). Then, for each of the identified leaf tile $t \in \mathcal{T}_\mathsf{S}$, the function getSelectedObjectsFromTile returns the objects $t.\mathcal{E}_\mathsf{S}$ that overlap with the Selection clause of the query (*line* 3). Finally, the evaluateSelectionClause procedure returns the objects $\mathcal{O}_\mathsf{S}$ selected from the Selection clause and the leaf tiles $\mathcal{T}_\mathsf{S}$ (*line* 5).

Procedure 2: getTilesRequireFileAccess($\mathcal{T}_\mathsf{S}$, Q)

```
    Input: $\mathcal{T}_\mathsf{S}$: leaf tiles that overlap with the Selection clause of the query;   $Q$ $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$: query
    Output: $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$: leaf tiles that require file access
 1  forall $t \in \mathcal{T}_S$ do
 2      if $t \in \mathcal{T}_{\mathsf{S}_f}$ then                                                    //tile is fully-contained in S
 3          if $D \neq \varnothing$ or $F$ can not be evaluated using $t.\mathcal{M}$ then   //Filter and/or Analysis clause is included and can
              evaluated using $t.\mathcal{M}$
 4              $accessRequired \leftarrow$ true
 5      else                                                  //tile is partially-contained in S; i.e., $t \in \mathcal{T}_{\mathsf{S}_p}$
 6          if $D \neq \varnothing$ then                                                        //Details clause is included
 7              $accessRequired \leftarrow$ true                                 //access file for the $t.\mathcal{E}_\mathsf{S}$ objects in $t$
 8          else if $F = \varnothing$ and $N = \varnothing$ then                                     //no Filter & Analysis clauses
 9              $accessRequired \leftarrow$ false
10          else if $N$ and $F$ can be evaluated using $t.\mathcal{M}$ then //Filter and/or Analysis clause is included and can evaluated
              using $t.\mathcal{M}$
11              $accessRequired \leftarrow$ false
12          else
13              $accessRequired \leftarrow$ true

14      if $accessRequired$ is true then                          //we have to access the file for the objects $t.\mathcal{E}_\mathsf{S}$
15          insert $t$ into $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$

16  return $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$
```

4.3.4 Determining the Tiles that Require File Access

The getTilesRequireFileAccess (Proc. 2) determines the tiles for which we have to access the file and read the attributes values. File access is determined by the intersection between a tile and the query (fully/partially contained), the operations defined in the query, and the metadata stored in each tile.

Particularly, Procedure 2 for each tile $t \in \mathcal{T}_\mathsf{S}$, examines if the tile is partially/fully-contained in query (*line* 2), and if the operations defined in the query can be evaluated by tile's metadata (*lines* 2-13). The procedure returns the tiles for which a file access is required (*line* 16). In case of fully-contained tiles (*line* 2) we have to access the file if a Details clause is defined, or a Filter is included, and its condition can not be computed using metadata. On the other hand, if tile is partially-contained (*line* 5), in case that a Details clause is defined in the query (*line* 6), we always have to read from the file the values of the objects included in the Details clause. Also, we have to examine if the computations defined in the Analysis and Filter clauses can be evaluated using the metadata that are already available in each tile (*line* 10).

4.3.5 Incremental Index Adaptation

During query evaluation, we employ an *incremental index adaptation* policy AP, which adapts the index structure *based on the user interaction.* Particularly, the index adaptation is performed using a *tile splitting* method, in which the tiles are *incrementally* split into subtiles and construct tiles' hierarchies. For each new subtile, its metadata are computed.

The adaptTiles (Proc. 3) reorganizes objects in the index by splitting tiles into smaller ones, based on the adaptation policy AP. The procedure takes as input the

Procedure 3: adaptTiles($\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}, Q$)

```
Input: $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$: leaf tiles for which file access is required;  $Q$: query
Parameters: AP: adaptation policy
Output: $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$: tiles resulted from $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ after splitting

forall $t \in \mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ do
    if AP.splitRequired ($t$) = true then
        $\mathcal{T}_a \leftarrow$ AP.split ($t$)                //construct the subtiles $\mathcal{T}_a$ by splitting tile $t$
        AP.reorganizeObjectsInSplittedTiles ($\mathcal{T}_a$, $Q$)
    else
        $\mathcal{T}_a \leftarrow t$
    insert $\mathcal{T}_a$ into $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$
return $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$
```

Procedure 4: readFile($\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, $\mathcal{O}_{\mathsf{S}}$, Q, $\mathcal{F}$)

```
Input: $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$: tiles for which file access is required;   $\mathcal{O}_S$: objects included in Selection clause;
       $Q$ $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$: query;   $\mathcal{F}$: raw data file
Output: $\mathcal{V}_{\mathsf{F}_A}$, $\mathcal{V}_{\mathsf{D}}$, $\mathcal{V}_{\mathsf{N}_A}$, attributes values required for the Filter, Details & Analysis clause

forall $o_i$ included in tiles $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ with $o_i \in \mathcal{O}_S$ do
    access $\mathcal{F}$ at file offset $f_i$
    $\mathcal{V}_{\mathsf{F}_{A\,i}}$, $\mathcal{V}_{\mathsf{D}_i}$, $\mathcal{V}_{\mathsf{N}_{A\,i}}$, $\leftarrow$ read the $o_i$ attributes values that are required for the F, D and N clauses
    insert $\mathcal{V}_{\mathsf{F}_{A\,i}}$ into $\mathcal{V}_{\mathsf{F}_A}$;   insert $\mathcal{V}_{\mathsf{D}_i}$ into $\mathcal{V}_{\mathsf{D}}$;   insert $\mathcal{V}_{\mathsf{N}_{A\,i}}$ into $\mathcal{V}_{\mathsf{N}_A}$;
return $\mathcal{V}_{\mathsf{F}_A}$, $\mathcal{V}_{\mathsf{D}}$, $\mathcal{V}_{\mathsf{N}_A}$
```

set of tiles for which, access to the file is required $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, and returns a new set of tiles $\mathcal{T}'_{\mathsf{S}_{\mathcal{F}}}$, which is a super-set of $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, containing the subtiles created by the splitting as well as the tiles' hierarchies info. For each tile $t \in \mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ the procedure examines if t has to be split using the method splitRequired, and, if so, reorganizes the objects into the new tiles.

Note that, a tile may be split, only when a query overlaps with it. This restructuring attempts to maximize the number of tiles which are fully-contained in subsequent queries. Fully contained tiles may improve the performance, by reducing the I/Os operations needed for answering the query (more details are presented in Section 4.4).

In our baseline implementation for VALINOR, the splitRequired method defines a numeric threshold for the maximum number of objects that a tile should contain. In case that more objects are contained in the tile a split is performed. The split procedure in our baseline implements a Quadtree method. That is, each tile t overlapping with the query and containing more objects than the threshold is split into 4 equally sized subtiles.

4.3.6 File Access

The procedure getTilesRequireFileAccess (Proc. 2), identifies the leaf tiles $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$, for which we have to access the file $\mathcal{F}$ in order to evaluate the query. Here, we present the readFile (Proc. 4) which reads from file data for the objects included in the $\mathcal{T}_{\mathsf{S}_{\mathcal{F}}}$ tiles.

For each object o_i in which is selected from the Selection clause, and contained in a tile for which file access is required, we read from the file at the offset f_i (*lines* 1, 2) the attributes values required for the Filter, Details & Analysis clause (*line* 3).

One of the goals we try to achieve in the design of the index, is to reduce the cost of I/O operations. For that, we first store the file offset of each object and we start reading the file from this position to retrieve its attribute values. Second, exploiting the way that VALINOR constructs and stores the object entries, we are able to access the raw file in a sequential manner. The sequential file scan increases the number of I/Os over continuous disk blocks and improves the utilization of the look-ahead disk cache.

During the initialization phase, the object entries are appended into tiles entries as the file is parsed (Alg. 1). Implementing tile entries $t.\mathcal{E}$ as a list, the entries in each tile are sorted based on its file offset. That is, for each $t \in \mathcal{T}$, $\forall o_i, o_j \in t.\mathcal{M}$, with list positions $i < j$, we have that $o_i.f < o_j.f$. Hence, in the query evaluation, we identify the tiles $\mathcal{T}_{\mathcal{F}}$ for which we have to read the file (Alg. 2, *line* 2). Then, from the lists of object entries in $\mathcal{T}_{\mathcal{F}}$, we read the objects from lists following a *k-way merge* based on objects file offset. This way, object values are read by accessing the file in sequential order. Note that, in our experiments, the sequential access results in about 8 × faster I/O operations compared to accessing the file by reading objects on a tile basis (i.e., read the objects of tile t_i, then read the objects of tile t_k, etc.).

4.3.7 Aggregate Metadata Management

The metadata is used to improve the performance of queries with an Analysis and/or Filter clause, by reducing both I/O and computation cost.

After the adaptation of the tiles, the metadata handler MH, using the values retrieved from the file, recomputes and updates the metadata for the subtiles created by the adaptation process.

The updateMetadata procedure (Algo 2, *line* 7): (1) determines for which attributes to compute or update the metadata; (2) computes metadata; and (3) updates metadata in the hierarchies of the tiles in case of splitting.

The metadata stored in the tiles is determined by the metadata handler MH considering the functions that are used in the Analysis clauses of the query.For every tile, the metadata handler keeps a hash table with keys the column number of a non-axis column in the raw file. Each key is mapped to that tile's synopsis metadata for that non-axis column. If the Analysis clause of query requests bivariate statistics for two attributes (e.g., correlation or covariance), the metadata handler also keeps metadata pertaining to the pair of attributes.

4.3.8 Filter, Details & Analysis clauses Evaluation

In the general case, the *Filter clause* requires to retrieve from file the values $\mathcal{V}_{\mathsf{F}_A}$ of the attributes included in the Filter conditions (Alg. 2, *line* 5). Using the retrieved values $\mathcal{V}_{\mathsf{F}_A}$, the filter conditions are evaluated over the $\mathcal{O}_{\mathsf{S}}$ objects for filtering out the query objects $\mathcal{O}_{\mathsf{Q}}$. However, there are cases where the metadata (e.g., min, max) may be used to evaluate the filter conditions and avoid file access.

To evaluate the *Details clause*, we have to access the file, since in order to reduce the index size, we do not store attribute values other than the two axis attributes[1]. For the objects $\mathcal{O}_Q$ we retrieve the values $\mathcal{V}_{\mathsf{D}}$ of the attributes included in the Details clause (Alg. 2, *line* 5). Then, for each object of $\mathcal{O}_Q$ the details values $\mathcal{V}_{\mathsf{D}}$ are combined with the axis attribute values, resulting to the set of tuples $\mathcal{V}_{x,y,\mathsf{D}}$.

Finally, the *Analysis clause* is evaluated using: (1) the existing metadata of the fully-contained tiles; and (2) the values retrieved from the file, for the partially-contained tiles.

Note that, although both the Select and Filter clauses operate as traditional selection operations on the data (the Select clause is evaluated over the two axis attributes, whereas the Filter clause on the non-indexed attributes), we explicitly consider them as different operations in our query model in order to speed up visual exploration operations. Filtering on non-axis fields has an implicit benefit on the performance, in the case that metadata for this attribute exists (e.g., a user revisits a tile with the same filter condition).

We have a similar restriction on the expressiveness of our approach for the grouping operation. Grouping primarily targets the two axis attributes, i.e., aggregates are computed at the level of the tiles included in the query window, whereas grouping on a non-index attribute (e.g., average age by gender) is implicitly enabled via filtering operations (i.e., average age per tile filtering the gender). We are aware of this restriction, nevertheless our model is not a general-purpose query model but rather serves the needs of basic exploration operations (e.g., panning, zooming)

4.4 Advanced Methods for Index Management – Initialization & Adaptation

In this section, we present two methods for the initialization and adaptation of the index during query evaluation. One of the goals for improving the query performance is to reduce the costly file reads that are needed for answering the query. The Details, Filter and Analysis clauses of the query usually require access to the raw file to fetch the values for the extra attributes involved in these clauses. In order to handle these cases, we compute and store per tile aggregated metadata for the contained objects. A subsequent query overlapping with this tile may use the stored metadata and avoid accessing the file in order to evaluate the query.

What makes possible for a query to exploit the metadata depends on whether the overlapping tile is *fully* or *partially* contained in the query; i.e., all of its objects are needed for answering the query or a subset of. In a *partially-contained tile* t, we have to: (1) traverse the objects in t in order to find the objects $t.\mathcal{E}_{\mathsf{S}}$ that are included in the Selection clause of the query; and (2) access the file in order to compute the metadata for $t.\mathcal{E}_{\mathsf{S}}$ objects. On the other hand, for a *fully-contained tile* t, there is no need to perform any of the aforementioned operations as (1) the required metadata have already been computed for t; and (2) there is no need to iterate over the objects in t to find the ones that are included in the window. As a result, we neither have to access the file for any of the object contained in t (i.e.,

[1] Note that, for both Filter and Details clauses evaluation, we can avoid file accesses by storing values for attributes other than axis. However, here we describe the setting which requires the minimum memory resources.

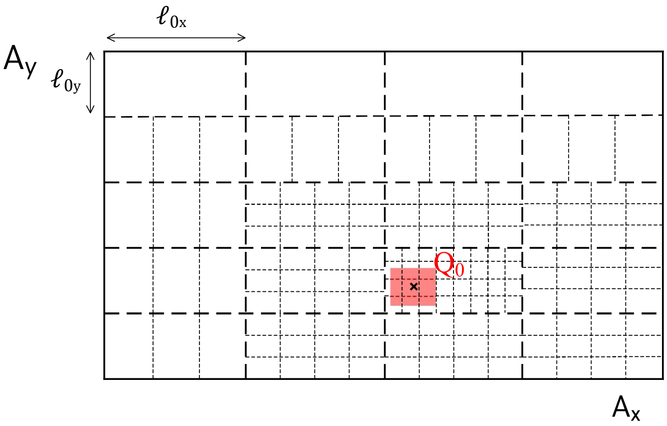


Figure 4.3: Query-Driven Index Initialization

I/Os cost), nor identify $t.\mathcal{E}_S$ (i.e., computation cost). Hence, fully-contained tiles reduce both computation and I/Os cost (for more details see Sect. 4.4.3).

In what follows we present our techniques, which aim to increase the number of fully contained tiles in a user exploration scenario by adjusting the initial tile structure, as well as incrementally performing index reorganization and metadata computations during query processing.

4.4.1 Query-driven Index Initialization

This section presents an advanced approach for the initialization of the VALINOR index. In our baseline initialization policy, we group objects into equal-size tiles but do not take into account the location of the initial user query in the 2D space as well as any subsequent user exploration actions for building the initial sizes of the tiles.

Assume that the user starts with an *initial query* Q_0, with (x_c, y_c) being the *center* of the *Selection clause*, *lying in the tile* t_0 and continues the exploration by applying the set of visual operations presented in Ch. 3. Recall that only the *move* and *zoom* operations change the visualized area to a new range; thus, subsequent queries corresponding to user operations performed at the early stages of the user exploration (i.e., user session) are *highly likely to reside (overlap) in tiles near to the initial tile* t_0.

To take advantage of this locality, VALINOR initializes tiles via a tile structure that is more fine-grained (i.e., having a large number of smaller tiles) in the area around the initial query. This is depicted in Figure 4.3, where given the first query, the size of initial tiles becomes larger as their distance from the initial query center (x_c, y_c) gets bigger.

Increasing the number of tiles near the first query, increases the possibility that subsequent user queries in this neighborhood overlap with fully-contained tiles, which in turn reduce the computation and I/O cost.

In what follows, we build upon the locality-based characteristic of the exploration model and propose a new approach, called *query-driven initialization policy*, for initializing the tiles of the index, based on the first user query and the potential next user actions. Note that the new method replaces the existing baseline initialization policy (*line* 2 of Algorithm 1) and is executed before the population of the tiles with object entries. At this stage, the query-driven initialization aims at *speeding* *up the initial actions* of the user session. When *combined with the adaptive splitting* (Sect. 4.4.2) the method provides fast results *for the entire user session.*

4.4.1.1 Query-driven Initialization Policy Overview

Our method considers that an initial set of tiles $\mathcal{T}_0$ is constructed for the index following the baseline equal-size initialization method, with each tile having a fixed size $\ell_{0_x} \times \ell_{0_y}$. Then, the Query-driven Initialization method takes as input: the constructed tiles T_0, the first user's query Q_0, and the number of extra tiles $\mathcal{T}_S$ it will create. For each tile $t \in \mathcal{T}_0$ the initialization method computes a numeric *initialization split factor* (SF). The SF factor determines the number of equally-sized subtiles which the tile t will be split into. In this case, the tile t will be the father tile of the new subtiles. For example, assume an initial query Q_0 and a tile $t \in \mathcal{T}_0$; then, if $SF_{Q_0}(t) = 4$, the tile t will be split into 4 equally-sized subtiles, with size of $\ell_{0_x}/2 \times \ell_{0_y}/2$.

4.4.1.2 Subtiles Size

Let $\mathcal{T}_0$ be the initial set of equally-sized tiles with area size $\ell_{0_x} \times \ell_{0_y}$ (i.e., $\forall t \in \mathcal{T}_0$, $|t.I_x| = \ell_{0_x}$, $|t.I_y| = \ell_{0_y}$); Q_0 is the initial user query with ranges $Q_0.I_x, Q_0.I_y$ and query center (x_c, y_c); and $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{T}_S$ is the set of tiles which the index will contain, with $\mathcal{T}_S$ being the subtiles created by splitting tiles in $\mathcal{T}_0$. The number of equally-sized subtiles, which a tile $t \in \mathcal{T}_0$ will be split into, is determined by its *initialization split factor (SF)*. SF is used for calculating the dimensions $\ell_x(t), \ell_y(t)$ of t's subtiles with respect to its initial dimensions ℓ_{0_x} and ℓ_{0_y}, as follows:

$$\ell_x(t) = \begin{cases} \ell_{0x}/\lfloor\sqrt{SF_{Q_0}(t)}\rfloor & \text{if } SF_{Q_0}(t) \geq 4 \\ \ell_{0x} & \textit{otherwise} \end{cases} \qquad \ell_y(t) = \begin{cases} \ell_{0y}/\lfloor\sqrt{SF_{Q_0}(t)}\rfloor & \text{if } SF_{Q_0}(t) \geq 4 \\ \ell_{0y} & \textit{otherwise} \end{cases}$$

Note that, splitting occurs only when $SF_{Q_0}(t) \geq 4$, i.e., $\sqrt{SF_{Q_0}(t)} \geq 2$; and the floor function is used for truncating the split factor to an integer value.

4.4.1.3 Initialization Split Factor (SF)

To compute the SF for a tile t, we model the likelihood that a subsequent query will overlap with t as a probability distribution over the distance of each point in t from the initial query Q_0 center (x_c, y_c), i.e.,

$$SF_{Q_0}(t) = \varrho_t \cdot |\mathcal{T}_S|$$

where, $\varrho_t = P(X \in t.I_x, Y \in t.I_y)$ is the *probability* that the next user query moves the query center within tile t. In other words, we treat X, Y as random variables corresponding to the center of a subsequent query performed by the user in the plane.

The formula distributes a fixed number of new subtitles to the initial set of tiles based on a probability distribution. The probability aims to adjust the splitting factor based on the distance of each initial tile from the initial query center. To achieve this locality-based splitting, the distribution of ϱ_t should decrease as the

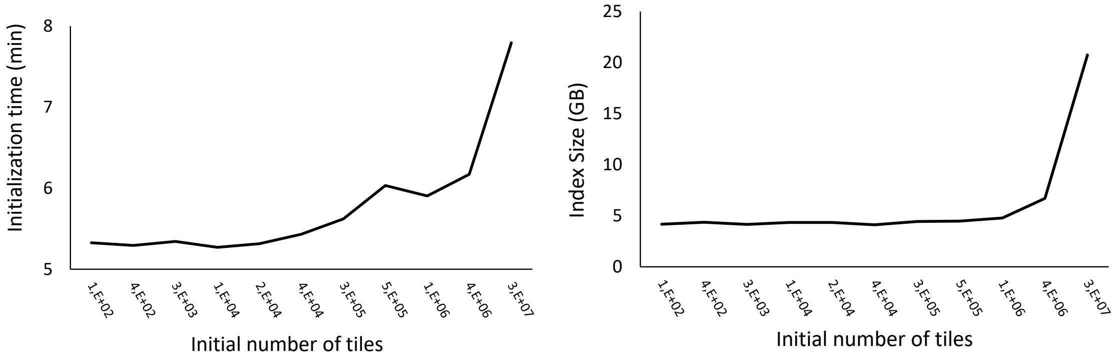


Figure 4.4: Initialization Time & Index Memory Size varying the Number of Initial Tiles

distance from (x_c, y_c) becomes larger. Although this probability can be computed using several factors, such as user moving patterns, visualization setting characteristics (e.g., screen size/resolution, visualization type) [52, 12, 89], we consider that it follows a bivariate normal distribution over the X, Y random variables; however, other distributions with similar characteristics could be considered. The probability density function is given by:

$$p(x,y) = \frac{\exp\left\{-\frac{1}{2}\left[\left(\frac{x-\mu_x}{\sigma_x}\right)^2 + \left(\frac{y-\mu_y}{\sigma_y}\right)^2\right]\right\}}{2\pi\sigma_x\sigma_y}$$

where, X, Y are independent (covariance is zero); $\mu_x = x_c$, $\mu_y = y_c$ (the initial query's center); and $\sigma_x = |Q_0.I_x|$, $\sigma_y = |Q_0.I_y|$, i.e., we set the standard deviation equal to the initial query range for the X and Y variables, respectively. The reason is that we require the majority of the new subtitles to be allocated in tiles at a distance of 3 query ranges from the initial query center.[2] This way we achieve a dense distribution around the initial query, entailing to smaller fully-contained tiles for the first user queries following the initial one.

4.4.1.4 Initialization Parameters Analysis

The initialization formula depends on the ℓ_{0_x}, ℓ_{0_y} ranges for the initial tiles $\mathcal{T}_0$, and the number of new subtiles $|\mathcal{T}_S|$ the index will create after the splitting.

We can express ℓ_{0_x}, ℓ_{0_y} at a scale of the overall exploration area, i.e., the ranges $|max - min|$ of the A_x and A_y attributes; i.e., $\ell_{0_x} = l \cdot |max(A_x) - min(A_x)|$ and $\ell_{0_y} = l \cdot |max(A_y) - min(A_y)|$, with $l \in (0, 1]$. Large values of l (the edge case of $l = 1$ is that initial range is the entire exploration area) result in a coarse-grained initial tile structure, especially for the areas far from the initial user session. The trade-off is that very large tiles are less likely to be fully-contained by subsequent queries, entailing an increased I/O and adaptation cost, when user moves to that area. On the other hand, too small l values increase the initial number of tiles even in locations far from the initial query, thus the memory and processing requirements of

[2] Recall that, according to the empirical 68-95-99.7 rule for the normal distribution, the 68% of the data is within 1 standard deviation (σ) of the mean (μ), 95% of the data is within 2 standard deviations (σ) of the mean (μ), and 99.7% of the data is within 3 standard deviations (σ) of the mean (μ).

the index. The edge case is creating more tiles than the number of objects, because for non-uniform datasets, there will be parts of the space with tiles containing no objects. Figure 4.4 presents the initialization time and index size in relation to the number of initial tiles for a synthetic dataset SYNTH10 (see Sect. 4.6). As can be seen, for larger numbers of initial tiles (i.e., smaller values of l) the initialization time and the memory requirements of the index increase. For example, for an initial number of tiles of 10K ($l = 1/100$) the initialization time and the index size are 5.27 min and 4.33 GB respectively, while for 25M tiles ($l = 1/5000$) it requires around 7.8 min and 21 GB.

In our experiments, we vary the l parameter with respect to the A_x and A_y ranges for several datasets with different distributions. From our study, we found that a value between 1/100 and 1/500 provides very good results for most of our datasets; i.e., the initial tiles of the equal-width methods is between 10K and 250K.

As previously mentioned, the above parameters can be estimated based on large number of factors, such as: visualization setting characteristics (e.g., screen size/resolution), visualization type, user moving patterns [52, 12, 89, 18]. However, this is beyond the scope of this work.

4.4.1.5 Memory Space Analysis

An upper bound of the total number $\mathcal{T}$ of tiles allocated during the initialization can be determined based on the memory constraints of the environment, as follows. Let $mem(t)$ be the footprint of each tile entry in memory, such that $mem(t) = b_t + b_o \cdot |t.\mathcal{E}|$, where b_t is a fixed number of bytes allocated for each tile record for holding its 2 ranges (e.g., 4 floats), initially computed metadata (e.g., 1 float) and a list of references (integers) to its children (if it is a non-leaf tile); b_o is a constant value for each object entry in the tile, keeping the A_x, and A_y values (e.g., 2 doubles) and its offset (e.g., a big int) from the beginning of file. The initial index memory footprint (before splitting) for $\mathcal{T}_0$ tiles is $mem(\mathcal{T}_0) = \sum_{t \in \mathcal{T}_0} mem(t) = |\mathcal{T}_0| \cdot b_t + b_o \cdot |\mathcal{O}|$, whereas after splitting the index footprint becomes $mem(\mathcal{T}) = \sum_{t \in \mathcal{T}} mem(t) = |\mathcal{T}_0| \cdot b_t + |\mathcal{T}_S| \cdot b_t + b_o \cdot |\mathcal{O}|$, as all object entries are contained in leaf nodes, thus considered only once in the memory allocation. Let mem_{MAX} be the maximum memory to be reserved for the initialization of the index, then $mem(\mathcal{T}) \leq mem_{MAX}$; i.e., $|\mathcal{T}_S| \leq (mem_{MAX} - |\mathcal{T}_0| \cdot b_t - b_o \cdot |\mathcal{O}|)/b_t$.

Note that, as $|\mathcal{O}| \gg |\mathcal{T}|$, the memory requirement for the index is heavily determined by the number of objects in the raw file. Also, the index size *is not affected by the number of attributes* comprising a record in the file as the VALINOR stores only the two attributes A_x, A_y of each object. In Section 4.5, we provide an eviction method for handling cases wherethe size of objects in memory do not fit in the allocated memory resources.

4.4.2 Query-driven Index Adaptation

In this section we present a method, called *Query-driven Tile Splitting* for restructuring the index based on the query window posed by the user. Particularly, this method implements the split function (line 3) of the adaptTiles procedure in Section 4.3.5. As presented, in Section 4.3, tiles visited by the query can be split into smaller ones, i.e., *the index is incrementally adapted to the user's interaction.* The index adaptation performs tile splitting, computes metadata and reorganizes objects into smaller groups during user exploration. The smaller tiles may result in larger numbers of fully-contained tiles during user exploration. The metadata of fully-contained tiles are going to be exploited by the next queries to reduce both I/O and computation cost. The basic characteristics of our adaptation method is that: (1) it follows a tile splitting process, where tiles split into subtiles, building tiles hierarchies; and (2) the subtile ranges are determined by the query ranges. The proposed method allows to perform the adaptation (compute the metadata, construct subtiles and reassign objects) without performing any extra I/O operations except the ones required for the query evaluation.

The baseline method we consider for VALINOR splits a tile that overlaps with the query to equally sized sub-tiles (Quadtree like). The main drawback of this method, is that in many cases where the split is performed, however, no metadata is computed for any of the constructed subtiles. Hence, the I/O that are performed during the query evaluation is not used anywhere. This occurs when the subtiles constructed by the splitting are not fully-contained in the query. On the other hand, in our query-driven splitting method, all the performed I/O operations are exploited to compute the metadata of subtiles. In what follows we outline the basic idea of our Query-driven Tile Splitting method.

4.4.2.1 Query-driven Tile Splitting Overview

We consider a query which contains an Analysis clause; i.e., non-axis attributes data is required for the query evaluation. Recall that during evaluation, for each partially-contained tile t, we access the file, and, for each object in the 2D area R^{S}_t that overlaps with the query, we retrieve the attribute values that are required for the Analysis clause. Then, we have to compute the metadata for the area R^{S}_t, for these objects.

Our method, during the processing of a query Q, splits t into subtiles, such that one of them t' corresponds to the R^{S}_t area. The metadata for the tile t' is computed during the evaluation of Q. Hence, in the case where one of the subsequent queries fully contains t', there is no need to access the file in order to compute metadata for this part of the query. The basic idea is better illustrated in the next example.

Example 5. [***VALINOR Adaptation & Query Processing***] Considering Example 4, after evaluating the query Q and adapting the index (Fig. 4.2), a subsequent query Q' is performed, as presented in Figure 4.5. We observe that the query Q' overlaps with the tiles $t_{2_a}, t_{2_b}, t_{2_c}, t_{2_d}$. Similarly to Example 4, in order to evaluate Q', we have to examine the overlapping tiles and identify the selected objects; also, for these tiles we have to determine for which of them we have to access the file.

We observe that the tile t_{2_c}, constructed during Q evaluation, is now fully-contained in Q' and its metadata has been already computed. Hence, for Q' evaluation we do not have to access the file for the objects o_1 and o_2 included in t_{2_c}.

For the evaluation of Q', we access only o_5, whereas in the case that no splitting occurs we had to access o_1, o_2 and o_5. Similarly to Example 4, during Q' evaluation, the tile t_{2_b} is further split in two subtiles $t_{2_{ba}}$ and $t_{2_{bb}}$. The tiles t_{2_a} and t_{2_d} are ignored since they do not contain any objects. Note, that the tile t_{2c} is fully-contained to the query and its metadata has been previously computed. So, in case that t_{2c} is split in this step, we have to compute metadata for the resulted subtiles. As result, we have to perform extra I/O operations to access the values of o_1 and o_2 from the

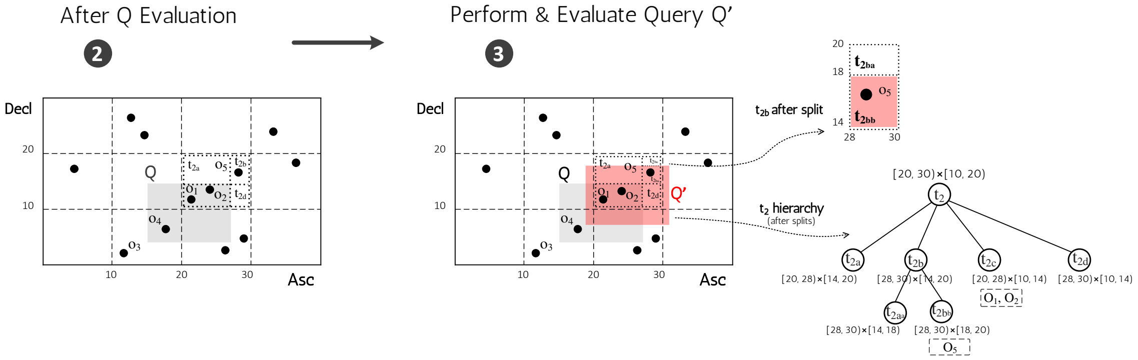


Figure 4.5: Index Adaptation and Query Processing

file.

4.4.2.2 Tile Splitting & Subtiles Construction

In our approach, each tile t of the partially-contained tiles $\mathcal{T}_{Q_p}$ is split into a set of disjointed subtiles. The subtiles are created based on the area R^{S}_t which captures the area that the query Q overlaps with t. Particularly, one of the new subtiles of t, denoted as *Query Subtile* t_Q, corresponds to the area R^{S}_t. In Figure 4.5, at the left, the *query subtile* corresponds to t_{2_c}.

Here, for ease of presentation, given a query Q, the intervals of the Selection clause $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$ are denoted as Q_x and Q_y, respectively. Given a tile t, the intervals of the tile $t.I_x$ and $t.I_y$ are denoted as t_x and t_y and we assume closed intervals for tiles. In what follows, we refer that an interval $I = [a, b]$ *is contained into an interval* $I' = [c, d]$, denoted as $I \subseteq I'$, when $a \geq c$ and $b \leq d$. Otherwise, I *is not contained* in I', denoted as $I \nsubseteq I'$. Further, we assume that the tile t with $t_x = [t_{x1}, t_{x2}]$ and $t_y = [t_{y1}, t_{y2}]$, is *partially-contained* in the Selection clause of the query Q with $Q_x = [Q_{x1}, Q_{x2}]$ and $Q_y = [Q_{y1}, Q_{y2}]$.

Based on the spatial relation between a partially-contained the tile t and the query Q, there are *four cases* based on which the subtiles are created. Figure 4.6 presents these four cases.

– **Case 1.** Case 1 holds when: (1) $t_x \subseteq Q_x$ and $Q_y \nsubseteq t_y$ and $t_y \nsubseteq Q_y$; or (2) $t_y \subseteq Q_y$ and $Q_x \nsubseteq t_x$ and $t_x \nsubseteq Q_x$. In the following definition and the Fig. 4.6, we assume the first condition. The second condition is also defined, in analogy.

In this case, *two subtiles* t_Q and t_a are constructed, where: ($\mathbf{t_Q}$) $t_{Qx} = [t_{x1}, t_{x2}]$, $t_{Qy} = [t_{y1}, Q_{y2}]$; ($\mathbf{t_a}$) $t_{ax} = [t_{x1}, t_{x2}]$, $t_{ay} = [Q_{y2}, t_{y2}]$.

– **Case 2.** In the Case 2, we construct *three subtiles*. This case holds when: (1) $t_x \subseteq Q_x$ and $Q_y \subseteq t_y$; or (2) $t_y \subseteq Q_y$ and $Q_x \subseteq t_y$. In the subtiles definition we assume the first condition (the case depicted in Fig. 4.6). In analogy, the second condition is defined.

In this case, *three subtiles* t_Q, t_a and t_b are constructed, where: ($\mathbf{t_Q}$) $t_{Qx} = [t_{x1}, t_{x2}]$, $t_{Qy} = [Q_{y1}, Q_{y2}]$; ($\mathbf{t_a}$) $t_{ax} = [t_{x1}, t_{x2}]$, $t_{ay} = [Q_{y2}, t_{y2}]$; ($\mathbf{t_b}$) $t_{bx} = [t_{x1}, t_{x2}]$, $t_{by} = [t_{y1}, Q_{y1}]$.

– **Case 3.** Case 3 holds when: (1) $t_x \nsubseteq Q_x$; and $Q_x \nsubseteq t_x$; and $t_y \nsubseteq Q_y$; and $Q_y \nsubseteq t_y$. In this case, *four subtiles* t_Q, t_a, t_b, and t_c are constructed, where: ($\mathbf{t_Q}$) $t_{Qx} = [Q_{x1}, t_{x2}]$,

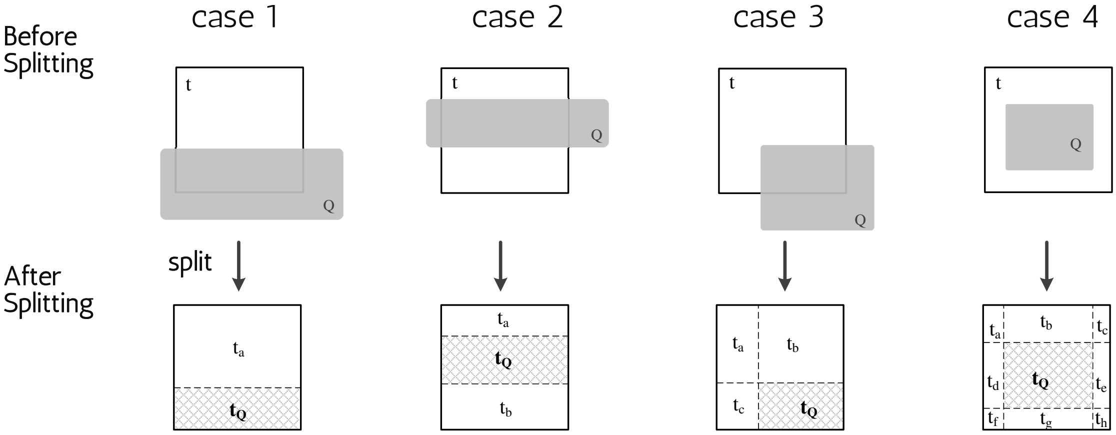


Figure 4.6: Tile Splitting Cases

$t_{Qy} = [t_{y1}, Q_{y2}]$; $(\mathbf{t_a})$ $t_{ax} = [t_{x1}, Q_{x1}]$, $t_{ay} = [Q_{y2}, t_{y2}]$; $(\mathbf{t_b})$ $t_{bx} = [Q_{x1}, t_{x2}]$, $t_{by} = [Q_{y2}, t_{y2}]$; $(\mathbf{t_c})$ $t_{cx} = [t_{x1}, Q_{x1}]$, $t_{cy} = [t_{y1}, Q_{y2}]$.

– **Case 4.** This case holds when: $t_x \subseteq Q_x$ and $t_y \subseteq Q_y$. In this case, *nine subtiles* t_Q, t_a, t_b, ... t_h are constructed, where: $(\mathbf{t_Q})$: $t_{Qx} = [Q_{x1}, Q_{x2}]$, $t_{Qy} = [Q_{y1}, Q_{y2}]$; $(\mathbf{t_a})$ $t_{ax} = [t_{x1}, Q_{x1}]$, $t_{ay} = [Q_{y2}, t_{x2}]$; $(\mathbf{t_b})$ $t_{bx} = [Q_{x1}, Q_{x2}]$, $t_{by} = [Q_{y2}, t_{y2}]$; $(\mathbf{t_c})$ $t_{cx} = [Q_{x2}, t_{x2}]$, $t_{cy} = [Q_{y2}, t_{y2}]$; $(\mathbf{t_d})$ $t_{dx} = [t_{x1}, Q_{x1}]$, $t_{dy} = [Q_{y1}, Q_{y2}]$; $(\mathbf{t_e})$ $t_{ex} = [Q_{x2}, t_{x2}]$, $t_{ey} = [Q_{y1}, Q_{y2}]$; $(\mathbf{t_f})$ $t_{fx} = [t_{x1}, Q_{x1}]$, $t_{fy} = [t_{y1}, Q_{y1}]$; $(\mathbf{t_g})$ $t_{gx} = [Q_{x1}, Q_{x2}]$, $t_{gy} = [t_{y1}, Q_{y1}]$; $(\mathbf{t_h})$ $t_{hx} = [Q_{x2}, t_{x2}]$, $t_{hy} = [t_{y1}, Q_{y1}]$.

4.4.3 Splitting Model Analysis

In this section, we analyze the cost of query evaluation via our splitting approach.

4.4.3.1 I/O Cost

We assume that the cost for reads is the same as the cost of writes, as c_{io} we denote the *I/O cost*, which is the *cost for reading/writing one object entry from/to the disk.*

4.4.3.2 Cost for Evaluating a Fully & Partially-contained Tile

The cost for a query is different when it is evaluated over a partially or a fully contained tile. Assume that a tile t is partially-contained in a query Q, with R_t^{S} to be the overlapped area. Recall that, $t.\mathcal{E}$ are the objects included in t; $t.\mathcal{E}_{\mathsf{S}}$ are the objects of t selected by Q (i.e., the objects included in the overlapped area R_t^{S}); and c_{io} be the cost of one I/O operation. Thus, the *cost* $C_{part}^{Q}(t)$ *of the evaluation of a query over a partially-contained tile* t is:

$$C_{part}^{Q}(t) = t.\mathcal{E} + c_{io} \cdot t.\mathcal{E}_{\mathsf{S}} \tag{4.1}$$

The $t.\mathcal{E}$ is the cost of scanning the objects $t.\mathcal{E}$ included in t in order to identify the objects $t.\mathcal{E}_{\mathsf{S}}$ that are included in the Selection clause S of the query. This is the cost of getSelectedObjectsFromTile function described in Section 4.3.3. Then, for each of the $t.\mathcal{E}_{\mathsf{S}}$ objects we have to access the file, and the cost is $c_{io} \cdot t.\mathcal{E}_{\mathsf{S}}$.

On the other hand, if t is fully-contained in a query Q, then $t.\mathcal{E} = t.\mathcal{E}_{\mathsf{S}}$; thus there is no need to scan every single object in t to assess whether it should be selected by the query nor to access the file for computing metadata for the tile (we assume that metadata is already computed by a previous query). Hence, the *cost* $C^{Q}_{full}(t)$ *of the evaluation of a query over a fully-contained tile* t is:

$$C^{Q}_{full}(t) = 0 \tag{4.2}$$

4.4.3.3 Splitting & Subtiles Construction Cost

The overall cost of splitting consists of the cost of splitting the tile t, constructing its subtitles, and reallocating the object entries of t in the new subtiles.

First, we have to determine the intervals of each subtile of t, and in the same time we define the subtiles as child tiles of t (i.e., initialize the child pointers). These can be performed without a cost, since the intervals of the subtiles are directly determined by the query select area R^{S}_t (Sect. 4.4.2.1). Then, we have to assign the objects $t.\mathcal{E}$ of the tile to the new subtiles. In the worst case 9 subtiles will be constructed (Case 4, Sect.4.4.2.1). Therefore, *the cost for splitting a tile* t is: $9 \cdot t.\mathcal{E}$.

4.4.3.4 Evaluation Cost in case of Splitting and not Splitting

Here, we are going to study, the improvement gained by performing a split. This analysis is going to be used in order to define the criterion for performing a split or not.

Assume a query Q that partially contains a tile t, and thus t is split based on our method resulting in a set of disjoint subtiles, one of which matches the query overlapping area, denoted as t_Q. Then, assume that the next query Q', partial contains t and fully contains t_Q.[3] Note that, this is a very common case in exploration scenarios, since as previously analyzed the user tends to explore nearby areas.

Next, we examine the cost for evaluating Q', in case of performing and not performing a split during the Q evaluation.

In case of *no split*, we have that Q' partially contains t. Thus, based on Eq. 4.1 the *evaluation cost* $\Phi^{Q,Q'}_{nosplit}$ of Q' *in case of no split*:

$$\Phi^{Q,Q'}_{nosplit} = C^{Q'}_{part}(t) = t.\mathcal{E} + c_{io} \cdot t.\mathcal{E}_{\mathsf{S}'} \tag{4.3}$$

In case of a *split*, Q' partially contains t and fully contains t_Q. In order to determine the evaluation cost in case of splitting we consider: the cost to evaluate the fully and partially-contained tiles (Eq. 4.1, 4.2); the tile's splitting cost $(9 \cdot t.\mathcal{E})$; and the cost to access the child tiles of t, which in worst case, we have to traverse 9 child pointers of t. Therefore, the *evaluation cost* $\Phi^{Q,Q'}_{split}$ for Q' *in case of split* is:

$$\Phi^{Q,Q'}_{split} = (t.\mathcal{E} - t.\mathcal{E}_{\mathsf{S}}) + c_{io} \cdot (t.\mathcal{E}_{\mathsf{S}'} - t.\mathcal{E}_{\mathsf{S}}) + (9 \cdot t.\mathcal{E}) + 9 \tag{4.4}$$

[3] The assumption that Q' is the next query, can be generalized to considering that Q' is one of the following queries (not strictly the next), if we consider that the tile t is not further split after Q.

4.4.3.5 Expected Splitting Gain

We use the costs $C^{Q,Q'}_{nosplit}$ and $C^{Q,Q'}_{split}$ of evaluating Q' in the two cases of not splitting and splitting, respectively. We define the *expected splitting gain* as the improvement in the performance of evaluating Q' in case of splitting the tile t during Q evaluation. Hence, based on the Eq. 4.3 & 4.4, the *expected splitting gain* $\Delta\Phi_{Q'}$ for the query Q' is defined as:

$$\Delta\Phi_{Q,Q'} = \Phi^{Q,Q'}_{nosplit} - \Phi^{Q,Q'}_{split} = c_{io} \cdot t.\mathcal{E}_{\mathsf{S}} \tag{4.5}$$

The final part of the equation results by omitting the cost of memory-based operations (i.e., tile's object scanning and splitting cost), since the cost of these operations is clearly dominated by the c_{io} cost of I/O operations.

4.4.3.6 Splitting Criterion: To Split, or not to Split?

We use the expected splitting gain as a criterion to determine, during the query evaluation, whether to perform a split or not. This gain is only an approximation indication, since it indicates the improvement over a single query when splitting is performed, without however taking into account future splits and queries. Otherwise, at an exhaustive scenario, we have to enumerate all possible queries and splitting scenarios which is prohibited in our online setting.

Let a numeric *splitting threshold* $\epsilon \in \mathbb{R}^+$. Using the expected splitting gain $\Delta\Phi_{Q,Q'}$ and the splitting threshold ϵ, we define a *splitting criterion*, in which a splitting is performed only when $\Delta\Phi_{Q,Q'} > \epsilon$. Hence, based on Eq. 4.5 we have:[4]

$$\textit{Splitting Criterion}: \text{ if } \left(c_{io} \cdot t.\mathcal{E}_{\mathsf{S}}\right) > \epsilon\,, \quad \text{perform a split} \tag{4.6}$$

We can observe in Eq. 4.6 that the criterion is defined based on I/O cost c_{io} and the objects $t.\mathcal{E}_{\mathsf{S}}$ of the tile t, selected by the query Q. These objects are computed during the Q evaluation; hence, defining the I/O cost, we are able to compute the splitting criterion on-the-fly during the evaluation of the Q.

4.5 Operating VALINOR Index under Memory Constraints

There are cases where the size of the index exceeds the memory available for its operation and parts of the structure have to be stored at the disk. Here, we define the *eviction policy* that determines which parts of the index are removed from main memory and written to the disk.

4.5.1 Preliminaries

4.5.1.1 Disk Storage Model

The eviction policy used in VALINOR is defined at the "tile-level". Whenever a tile is evicted from main memory, all its records are removed from main memory

[4] The threshold ϵ can be determined based on numerous factors such: hardware performance, tiles and query sizes, etc. However, this is beyond the scope of this work.

and written to disk, or conversely, read from disk to memory (i.e., fetched) when we retrieve it for usage. Note that, the "tile-level" policy described here can be easily adapted to accommodate a "record-level" policy, in which individual records from tiles can be selectively evicted and stored in disk.

Each time a tile is selected to be evicted, all of its objects *currently residing in memory* are written to the disk[5] The objects of a tile may be written to different positions in the disk (i.e., organized in different files) and a pointer attached to the tile indicates the tile's position in the disk. The use of different files allows to store the objects of each tile in sequential manner. In our disk storage model, we denote as $N = |\mathcal{O}|$ the number of objects in the dataset. Further, we assume that the main memory can fit M objects[6], with $N > M$.[7]

4.5.1.2 Eviction Phases

The objects' evictions are performed in two different phases. The first is during the *index initialization* phase, and the second is during *query processing.* Recall that eviction is performed only when the size of objects in tiles exceeds the memory size.

During the *index initialization* and while reading the objects from the source file and building the index, if the memory gets full, we evict tiles (and write them in disk), in order to free memory up and read the remaining objects. Recall that, during initialization all objects must be read from the source file and indexed.

During *query processing*, a query may overlap with tiles which have been evicted and stored in the disk. In that case, we first have to free memory and then fetch previously evicted tiles needed by the query; i.e., first we write "memory-based" tiles to the disk, and then we fetch the evicted tiles from the disk into memory. In what follows, we describe the eviction during the two phases.

4.5.2 Eviction During Query Processing

An eviction is performed when a query overlaps with a tile which has been previously written to disk. In that case, in order to fetch the required tile, we have to free memory by writing another tile to the disk. Before we define the eviction policy, we present some necessary definitions.

Tile Disk Access Cost. Each *tile* t is associated with a *disk cost* $C_{io}(t)$ that is the cost of reading/writing the objects entries $t.\mathcal{E}$ from/to the disk. Recall that, we assume that the cost for reads is the same as the cost of writes, as c_{io} we denote the *cost for reading/writing one object entry from/to the disk* (Sect. 4.4.3). The *tile disk cost* $C_{io}(t)$ for tile t is the cost of reading/writing all objects of t from/to disk. That is, $C_{io}(t) = c_{io} \cdot |t.\mathcal{E}|$. Note that, the cost $C_{io}(t)$ is imposed in both cases where: (1) the eviction policy selects to write a tile t to the disk; and (2) a query accesses a tile t, which is stored in the disk.

[5] Tile's metadata will also be written to the disk, however here for simplicity we assume that there are no metadata stored in tiles.

[6] Section 4.4.1 presents the memory requirements of a tile and an object.

[7] Note that, here for simplicity, we assume that M has be calculated by excluding from "actual" memory size, the memory required to store the information related to the index structure; e.g., tiles intervals, pointers, etc.

Tile Eviction Score. A tile t is associated with a numeric *eviction score* $t_{evSc} \in [0, 1]$, which formulates the possibility that the tile t is going to be selected by (i.e., overlapped with) a next query. The highest is the score, the more likely is for the tile to be selected by a subsequent query. This score can be computed considering several factors, such as: the size of the tile's area w.r.t. query' selection area size; temporal and spatial locality of the tile w.r.t. previously expressed queries; user moving patterns, visualization type, screen size/resolution [52, 12, 89, 18]. However, this is beyond the scope of this work. In our implementation, considering the "locality" of exploration scenarios, we define the eviction score based on the Euclidean distance between the tile and the query.

Expected Eviction Cost. The *expected eviction cost* $\mathbb{E}_t$ for a tile t combines (1) the tile disk access cost $C_{io}(t)$; and (2) the eviction score t_{evSc} of t, as

$$\mathbb{E}_t = t_{evSc} \cdot C_{io}(t) \tag{4.7}$$

The overall expected eviction cost for a set of tiles $\mathcal{T}_e$, is computed as the sum of the costs of all tiles. That is, $\mathbb{E}_{\mathcal{T}_e} = \sum_{\forall t_i \in \mathcal{T}_e} \mathbb{E}_{t_i}$. Obviously (also in our implementation) one can consider $C_{io}(t)$ to be constant, especially, if all accesses are at the same disk.

Based on the aforementioned definitions, in what follows, we formulate eviction policy that is adopted during query processing.

Eviction Policy. Let V be the number of objects, which have to be evicted from memory. The eviction policy selects the tiles $\mathcal{T}_e$ to be evicted, such as the overall expected eviction cost $\mathbb{E}_{\mathcal{T}_e}$ of $\mathcal{T}_e$ is minimized and the tiles of $\mathcal{T}_e$ contains at least V objects. Hence, formally we have:

$$\text{minimize} \sum_{\forall t_i \in \mathcal{T}_e} \mathbb{E}_{t_i} \quad \text{subject to} \sum_{\forall t_i \in \mathcal{T}_e} |t_i.\mathcal{E}| \geq V \tag{4.8}$$

Selecting Tiles to be Evicted. Considering the objective of the eviction policy (Eq. 4.8), we adopt a generally known approximation approach to select the tiles that are going to be evicted. Initially, we sort the tiles based on their expected eviction cost $\mathbb{E}_t$, in descending order. Then, we select and evict the top tiles which in sum contain at least V objects.

Reconstruction If, during query processing, a tile that has been evicted overlaps a query and we need to examine its objects, we fetch the objects that are in the disk and we merge with the ones in memory to recreate the complete list of a tile's objects. Note that during this recreation, the list of objects preserves its original order of insertion. To minimize the associated I/O costs, during fetching, no objects are erased from the disk. In this way, if a tile needs to be evicted again, we remove its objects from memory and only write to the disk the ones that were not written before.

Table 4.2: VALINOR Evaluation: Datasets

Name	Num of Object	Num of Attributes	Data Size (GB)
Real Datasets			
SDSS	40M	446	270
TAXI	165M	18	26
Synthetic Datasets			
SYNTH10	100M	10	11
SYNTH50	100M	50	51

4.5.3 Eviction during the Initialization Phase

In this section we describe the eviction method that is followed during the initialization phase. As already mentioned, we adopt a "Tile-level" eviction method. During the initialization phase, new records read from the source file are placed into tiles. If the main memory gets full as we read the objects from the source file,we have to free memory by writing tiles to the disk in order to make space and read the remaining objects. "Tile-level" eviction means that all tile objects which have been read into memory up to the time eviction occurs are stored to disk, while the population of the tile continues. That means that an eviction may occur on a tile, when its current objects exceed the memory limitations and the tile may keep receiving new objects from the file parsing and store them in memory, after that last eviction. In this way, some of the objects of a tile may reside in the disk, and some may be in memory.

During the initialization phase, each time the memory gets full, the eviction policy selects the tile with the following two properties: (1) it has not been previously evicted, and, (2) it has the minimum eviction score among all candidates (specifically, this is the tile located far away from the initial query's range), and writes its objects currently residing in memory to disk.

4.6 Experimental Analysis

In this section, we conduct the experimental evaluation of VALINOR. We first present the experimental setup which describes the datasets, the evaluation scenario, the setting for the competitors and details about the implementation of VALINOR and present the results.

4.6.1 Experimental Setup

4.6.1.1 Datasets

We have used two *real datasets*, the *NYC Yellow Taxi Trip Records* (TAXI), which is a csv file, containing information regarding yellow taxi rides in NYC[8], and the *Sloan Digital Sky Survey dataset* (SDSS). From the TAXI dataset, we selected a subset that includes taxi trip records in 2014 (165M objects, 26 GB) with each record object referring to a specific taxi ride described by 18 *attributes* (e.g., pick-up and drop-off dates and locations, trip distances, fares, and tip amount). Table 4.2 presents the basic characteristics of the datasets. In our experiments for the TAXI dataset, the

[8] Available at: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

pickup location longitude and latitude were selected as the axis attributes, and the two attributes for which statistics were calculated were the trip distance and the tip amount. Each query is defined over an area of 500m × 500m size (i.e., window size), simulating a map-based exploration at the neighborhood zoom level, with the first query Q_0 posed in central Manhattan (a very dense area).

From the Sloan Digital Sky Survey dataset, we used in our experiments a csv file (270 GB) containing 40M rows of the the PhotoObjAll table, each row described by 446 *attributes*. The right ascension and declination attributes were selected as the axis attributes of our exploration scenario.

Regarding the *synthetic datasets* (SYNTH10/50), we have generated two csv files of 100M *data objects*, having 10 and 50 *attributes* (11 and 51 GB, respectively). Each attribute value is a real number in the range (0, 1000) and follows a uniform distribution. For the query sequences we generated for the *synthetic dataset*, we used a window size with approximately 90K objects.

4.6.1.2 Evaluation Scenarios

We study the following visual exploration scenario: (1) First, the user selects the two axis attributes and requests to explore a region of the data from the raw file, specifying also the attributes for which statistics will be calculated. For this action, referred to as "From-Raw Data-to-1stResults", we measure the *execution time* for creating the index and answering the first query, the results of which are evaluated directly on the raw file, during index initialization. (2) Next, the user continues exploring areas of the dataset.

User's Entry Point. For selecting the entry point (initial query Q_0) of the user we adopt the following. In the TAXI dataset, the position of Q_0 is defined over the NY Manhattan area. In the SYNTH10/50 datasets, the position of the initial query is randomly selected over the whole area. Finally, the SDSS dataset is very sparse, there are numerous, large empty areas (i.e., without containing objects), so we find a not-empty area to evaluate our queries.

Query Size. The initial size of the queries, for the TAXI dataset the size corresponds to one city block in the Manhattan. For the SYNTH10/50 and SDSS datasets, we follow an approach which is based on visualization-based assumptions. The maximum number of objects that can be visualized without having objects' overlaps (i.e., two objects are very close and appear as a single object) can be estimated assuming that: each can be visualized in one pixel, and there are no objects in the pixels around it.

In this setting, the maximum number of visualized objects is $(w \times h)/9$, where $w \times h$ is the resolution of the screen. Today the most common resolutions in desktops are 1366×768 and 1920×1080[9], which results in about 100K to 200K objects to be visualized. Therefore, the size of the queries in SYNTH10/50 contains about 100K objects, and in SDSS about 200K objects.

Exploration Scenarios. In our evaluation we examine two exploration scenarios. In the *first scenario*, we generated sequences of 100 overlapping queries, with each window query shifted in relation to its previous one by 1-20% towards a random

[9] `https://gs.statcounter.com/screen-resolution-stats/desktop/worldwide`

direction (N, E, S, W, NE, NW, SE, SW). This scenario attempts to formulate a common user's behavior in 2D visual exploration, where the user explores nearby regions using pan operations. [101, 102, 54, 89, 12, 96, 26, 29]. For example, assume the common "region-of-interest" or "following-a path" scenarios in map visual exploration.

The *second exploration scenario* combines pan and zoom operations. Particularly, based on the findings of [12] for 2D exploration, the users perform almost equal number of pan and zoom operations. Further studies [79] have shown that in general in map-based visual exploration tasks, the users change the zoom level at most 3 (i.e., +/- 3 levels w.r.t. zero level). Thus, in our second scenario, we assume that a user performs a pan or a zoom operation with equal probability. In case of pan, we follow the strategy used in the first scenario (i.e., random shift 1-20% toward a random direction). For the zoom operations, we consider that a user has equal probability of performing a zoom-in or a zoom-out operation. Each zoom-in/zoom-out operation increases or decreases the visualized area to 150% in relation to the previous one.

4.6.1.3 VALINOR Variations

To assess the effect of the initialization and adaptation policy, we measure the performance of three variations of VALINOR. In the first variation called VALINOR-S, we use the basic initialization mode without index adaptation. With this setting, VALINOR essentially works as a static flat-tile structure that does not adapt to the query workload. In the second variation, called VALINOR-B, we use our basic initialization mode with the basic quad-tree like adaptation mode, while in the third (VALINOR), we use the query-based initialization mode (Sect. 4.4.1) with the query-driven adaptation mode (Sect. 4.4.2). For every one of the variations, we initialized the index with $l = 1/100$ resulting in an initial grid of 100 × 100 equal-width tiles (this number of initial tiles is used in all the experiments). Also, we set the number of extra tiles $|\mathcal{T}_S|$ which will be created during the Query-driven initialization method to a 20% of the number $|\mathcal{T}_0|$ of initial tiles. Recall, these new tiles will be distributed around the first query Q_0. For both adaptation modes, we set the threshold for the number of objects required in order to split a tile equal to 200.

4.6.1.4 Competitors

We have compared with: (1) A traditional DBMS (MySQL 8.0.15), where the user has to load all data in advance in order to execute queries; three indexing settings are considered: (*a*) no indexing (SQL-0I); (*b*) one composite B-tree on the two axis attributes (SQL-1I); and (*c*) two single B-trees, one for each of the two axis attributes (SQL-2I). MySQL also supports SQL querying over external files (see CSV Storage Engine in Chapter 2); however, due to low performance [7], we do not consider it as a competitor in our evaluation[10]. (2) PostgresRaw (PostgresRaw)[11], build on top

[10] We refer the reader to [7], which has performed several experiments comparing the PostgresRaw against two DBMSs (MySQL and a commercial DBMS). The experiments demonstrated the (noticeable) poor performance of the DBMS systems against PostgresRaw (e.g., in some experiments PostgresRaw is about 12× faster than the MySQL), which is due to the fact that each time a query is posed to external data, the whole file needs to be parsed.

[11] https://github.com/HBPMedical/PostgresRAW

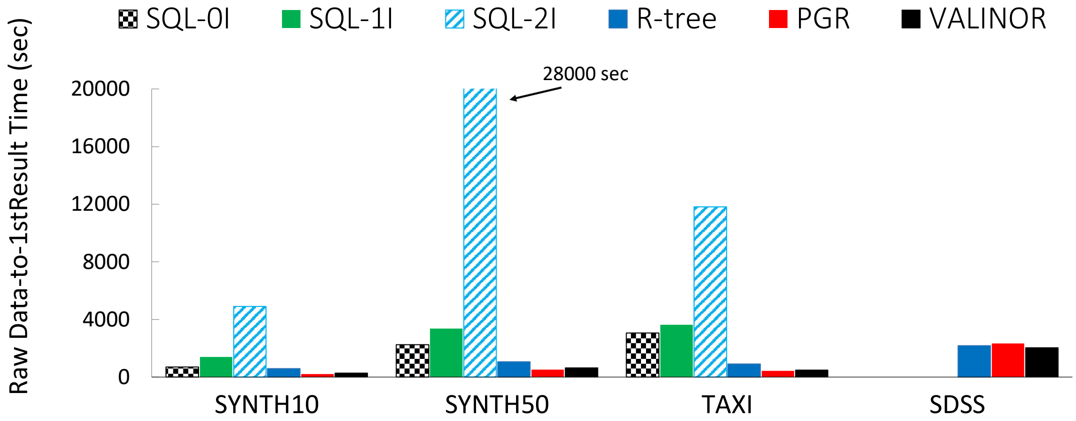


Figure 4.7: VALINOR: Time for Answering the 1st Query over the Raw File. Time includes: File Parsing, Index Construction & Q_0 Evaluation

of Postgres 9.0.0 [7], which is a generic platform for in-situ querying over raw data (Ch.. 2). (3) A main memory Java implementation of the R*-tree[12] [13]. We have tested various configurations for R-tree index fan-out, ranging from 4 to 128; as the difference in the performance is marginal, we only report on the best one, i.e., 16. For all the other tuning decisions, with respect to its performance and memory minimization, we have setup the R*-tree with the configuration recommended in its GitHub repository.

4.6.1.5 Metrics

We compare our method with the existing solutions, as well as with our baseline approach. We measure the: (1) *execution times* for each query in the sequence; (2) *accumulative execution time* for the entire exploration scenario; (3) *memory consumption*; (4) the performance of the *eviction mechanism* under varying memory constraints; and (5) the number of *I/O operations*. In all cases, the reported time values are the averages of 10 executions.

4.6.1.6 Implementation

We have implemented RawVis[13] on JVM 1.8 and the experiments were conducted on an 3.60GHz Intel Core i7-3820 with 64GB of RAM. We applied memory constraints (max Java heap size) in order to measure the performance of our approach and our competitors in a commodity hardware setting. For large datasets, PostgresRaw required a significant amount of memory (in some cases more than 32GB); the same held for the in-memory R-Tree implementation (>16GB in most cases). In contrast, VALINOR performed well in all datasets (>250GB) for heap size less than 10GB (see Sect. 4.6.2).

[12] https://github.com/davidmoten/rtree

[13] The source code is available at https://github.com/Ploigia/RawVis

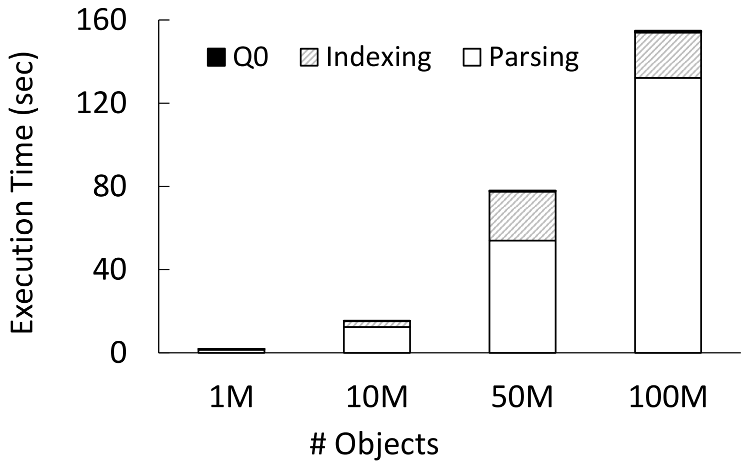


Figure 4.8: VALINOR Initialization Phase: File Parsing, Index Construction & Q_0 Evaluation

4.6.2 Results

4.6.2.1 From-Raw Data-to-1st Result Time

In this experiment, we measured the time required to answer the first query Q_0. This time includes the time required to load and index the data for MySQL, and to construct the positional map for PostgresRaw. For the VALINOR and R-tree cases the in-memory indexes must be built. For the R-tree construction, bulk-loading was used.

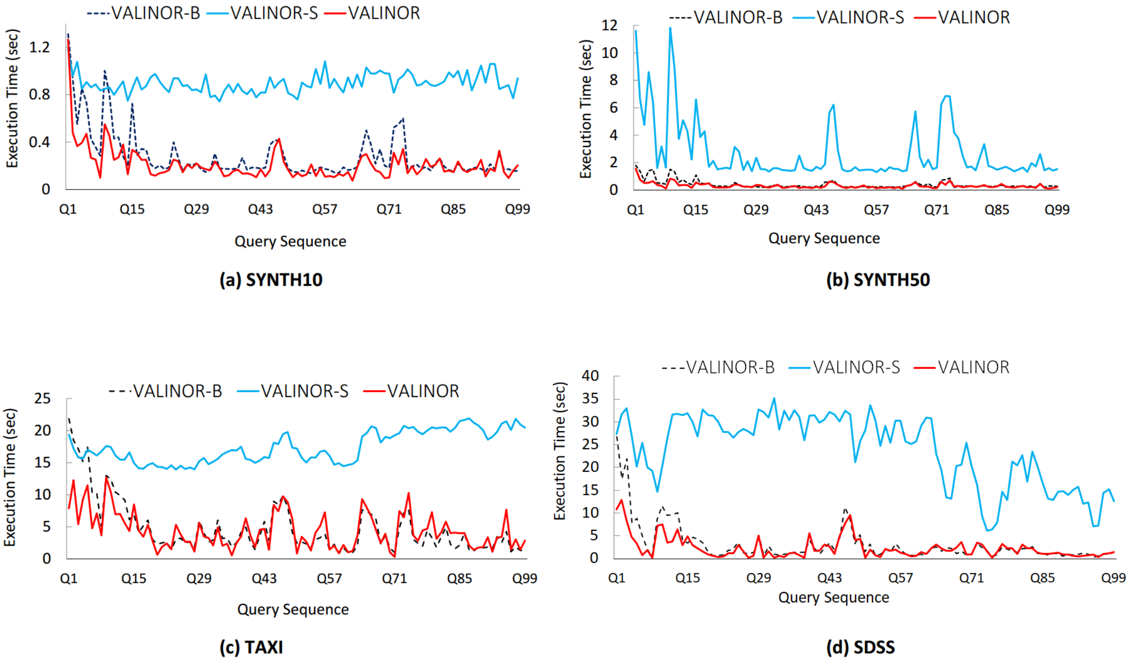
Figure 4.7 presents the results for the datasets used. In these results, we omit MySQL for the SDSS dataset as it took more than 5 hours just to load the dataset without creating any indexes. VALINOR outperforms the MySQL and R-tree methods, for all datasets. Before being able to answer the first query, MySQL needs to parse and convert all attributes of the raw file and store all data on disk. Also, for the SQL-1I and SQL-2I cases, the corresponding indexes must be built, which explains the increased initialization time in relation to SQL-0I where no index is generated.

Further, as expected, VALINOR exhibits a lower initialization time than R*-tree; the latter must determine multilayer MBRs and assign objects to leaf nodes as opposed to our approach which is initialized with fixed tile sizes.

In this experiment, VALINOR exhibits a slightly higher initialization time in relation to PostgresRaw for the SYNTH10/50 and TAXI datasets. This can be attributed to the non-optimized csv parsing and slower I/O Java operations, as opposed to the efficiency provided by the programming language of PostgresRaw (i.e., C) – of course, improving our implementation in terms of parsing and I/O is open for exploration in the future.

Despite this slight difference in initialization time, as demonstrated latter, VALINOR is considerable faster in answering queries during an exploration scenario. Particularly, *during exploration, in most cases, VALINOR is about 5-10× faster compared to existing systems.*

For the largest dataset (SDSS), which contains 446 attributes, VALINOR outperforms the other methods. Particularly, VALINOR populates the index only for the two axis attributes and stores tile metadata for the attributes requested in the



(a) SYNTH10 **(b) SYNTH50**

(c) TAXI **(d) SDSS**

Figure 4.9: VALINOR Initialization & Adaptation Methods: Execution Time Comparison of the 3 VALINOR Configurations

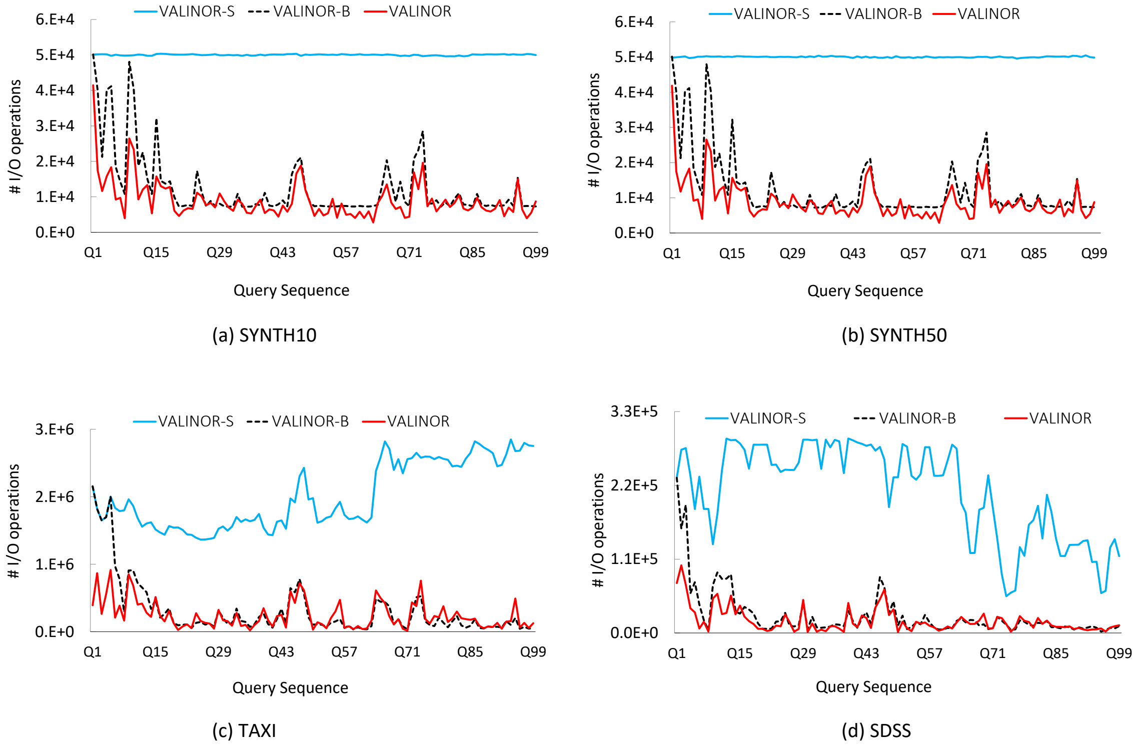
analysis clause of the queries. PostgresRaw, on the other hand, populates its index (positional map) with the position of all tokenized attributes until the last attribute requested in the query. For the queries posed in SDSS, this last attribute corresponds to the declination which is the 398th attribute in the dataset. As a result, PostgresRaw keeps in the positional map the position of the first 398 attributes, which explains the slower initialization time.

Finally, for assessing the time required for VALINOR for answering the 1st query Q_0, we have separately measured the time of the initialization phase that spent to: parse and read the file, construct the index and determine the objects of the first query Q_0. In our experiment we use the SYNTH10 data varying the number of objects from 1M to 100M objects. The results are presented in the Figure 4.8. In all cases, the time required for parsing the file clearly dominates (more than 70%) the overall initialization time. On the other hand, since the first query Q_0, is evaluated during the file parsing and the index construction, the query evaluation overhead is negligible.

4.6.2.2 Initialization & Adaptation Methods

Next, we evaluate the performance of the three VALINOR variations, and show that the query-driven initialization and adaptation policies improve query execution time, especially for the first operations of the exploration scenario. Figure 4.9 presents the execution time for queries $Q_1 \sim Q_{99}$. Note that Q_0 is not depicted in the figures. This query, which triggers the initialization of the index, is answered directly from the raw file and does not exhibit any significant difference among the VALINOR variations presented.

As we can observe, VALINOR-S exhibits the worst performance for all datasets. In VALINOR-S, there is no adaptation to the workload in order to increase the number of fully-contained tiles with precomputed aggregate values. Both, VALINOR-



(a) SYNTH10

(b) SYNTH50

(c) TAXI

(d) SDSS

Figure 4.10: VALINOR Initialization & Adaptation Methods: Number of I/O Operations
Comparison of the 3 VALINOR Configurations

B and VALINOR perform tile splitting to minimize future file reads, however in VALINOR, as can be seen, the query-driven initialization and adaptation policies used provide an initial boost in query performance. This boost is more significant for the TAXI and SDSS datasets, since the window size used for their workload is much smaller in relation to the initial tile size. In VALINOR, the query-driven initialization policy splits the area around the first query in a more fine-grained fashion, making subsequent neighboring queries fully overlap more tiles sooner and reducing their execution time. This initial boost in query performance is also the result of the query-driven adaptation policy employed. Using this adaptation method, the subtiles that correspond to the intersection with the query are more likely to fully overlap with similarly-sized subsequent queries. This is in contrast to the basic adaptation mode, where a tile may need to be split multiple times to create subtiles small enough to be fully contained by the next queries. Also, in the query-driven adaptation policy, we exploit all the I/O operations for the subtiles that correspond to the intersection with the query by computing metadata for them. As a result, in VALINOR, the index adapts to the workload and executes the first queries faster than in VALINOR-B. However, as can be seen in Figure 4.9, both adaptation methods manage to adapt to the workload and exhibit a similar performance after a number of queries (e.g., approximately after 15 to 20 queries). Note that, this behavior is aligned with the goals of the optimizations proposed in this work; i.e., to improve the overall response time, especially for the user operations performed at the early stages of the exploration scenario.

The execution time examined above, is mainly determined by the number of I/O operations required to answer each query. This is evident in Figure 4.10, where as it

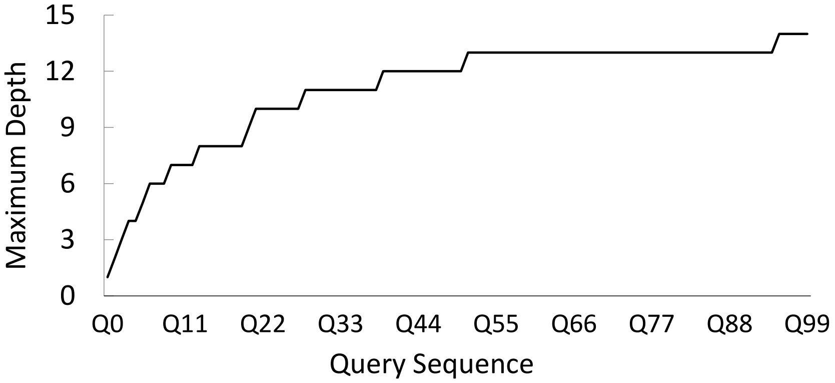
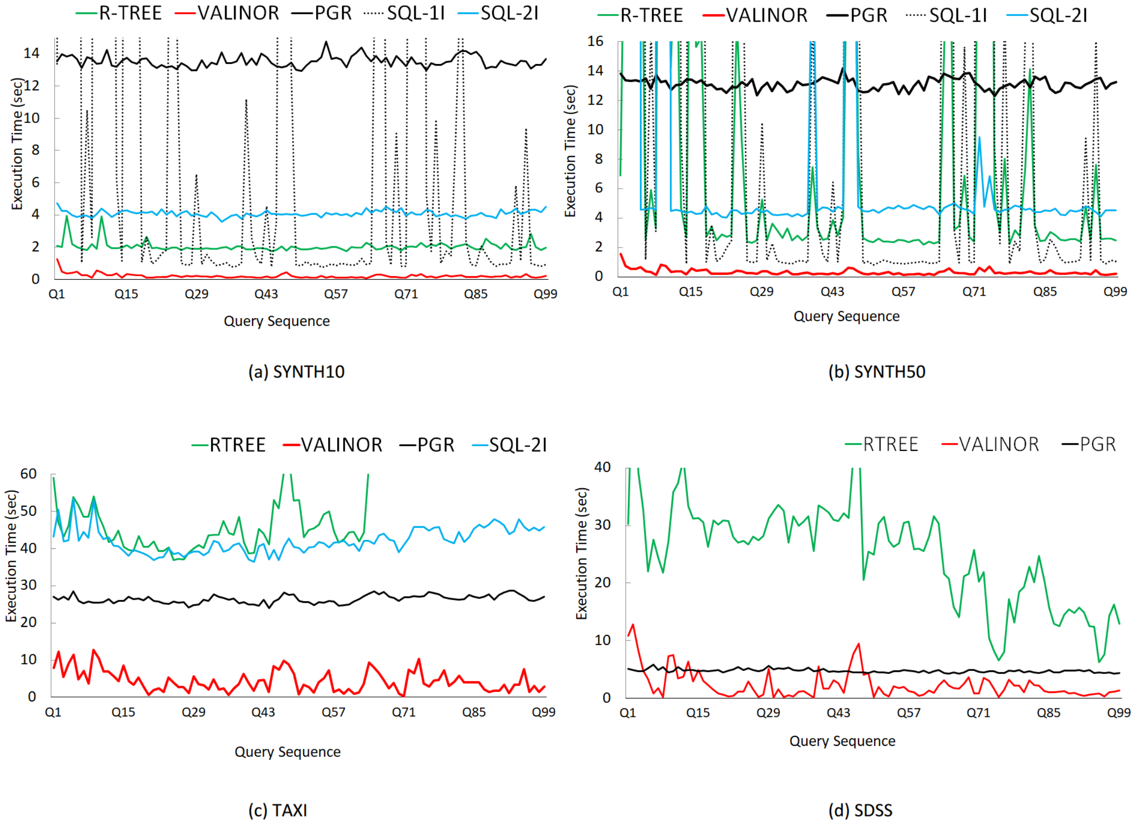


Figure 4.11: Maximum Hierarchy Depth of VALINOR per query (TAXI)

can be seen, the plots follow closely the corresponding execution time plots in Figure 4.9. Regarding the two synthetic datasets (SYNTH10/50), their I/O plots almost completely match (Fig. 4.10a, b). These two datasets have the same number of rows and all of their attributes have values uniformly distributed in the same range. Their only difference is the number of attributes each one has (i.e., 10 and 50 respectively). Thus, since we use the same query workload and the same initialization setting, the I/O operations required for both datasets are similar. Also, every query in their workload had the same window size and selected approximately the same number of objects. This explains why in VALINOR-S, where the index does not perform tile splitting in order to reduce the file accesses of subsequent queries, the number of I/O operations does not change from query to query. Overall, for the synthetic datasets, VALINOR requires around 30% less I/Os compared to VALINOR-B and 80% less compared to VALINOR-S; 22% and 87% for TAXI, 30% and 92% for SDSS respectively.

Regarding the index adjustment to the query selection predicate, the incremental index adaptation performs a larger number of tile splittings in areas that are frequently visited by the user. As a result, an unbalanced index is constructed, with deeper tile hierarchies in those areas. On the other hand, the threshold used by our splitting method (Sect. 4.4.3) limits the number of times a tile is split. Figure 4.11 presents the maximum depth of the index resulting from every query in the sequence for the TAXI dataset. The initial depth of the index after Q_0 is one. Between queries Q_1 ~ Q_{99} where the user explores neighboring areas, the query-driven adaptation method, further splits the tiles and increases the maximum depth of the index. We observe, however, that due to the threshold limit, the depth converges to a maximum value (14 for the TAXI dataset).

To assess the influence of Q_0 on how the index is refined during the entire exploration scenario, we have conducted an experiment in SYNTH10, in which we varied the initial query, while keeping constant the remaining workload of queries Q_1 ~ Q_{99}. Since this dataset has a uniform distribution, the position of Q_0 does not significantly affect the initial tiling of the index. Thus, we only varied the Q_0 size (from 0.01% to 10% selectivity on the dataset) and we measured the way the index is refined (number of total tiles) after every query. Figure 4.13 shows that although Q_0 size affects the initial tile structure, VALINOR attempts to adjust the number



(a) SYNTH10

(b) SYNTH50

(c) TAXI

(d) SDSS

Figure 4.12: Execution Time: VALINOR vs. Competitors

of tile splittings that happen after Q_0. For small Q_0 sizes, the index is already split in more small tiles around Q_0 and following queries create fewer tiles compared to larger sizes of Q_0. This explains why for larger Q_0 the number of total tiles increases more rapidly at first. Still, as can be seen in the figure, after Q_{85} the number of new tiles created by tile splittings are approximately the same despite different Q_0.

4.6.2.3 VALINOR vs. Competitors during Exploration Scenarios

In this experiment, we compare the behavior of VALINOR against the existing solutions. Figure 4.12 shows the execution time for queries $Q_1 \sim Q_{99}$, without the first query that includes the initialization stages for every system (e.g., loading and indexing the data for MySQL).

In the results, we omit the plots for SQL-0I for the two synthetic datasets, and the ones for SQL-0I and SQL-1I for the TAXI, as the corresponding execution times were much higher (more than 350sec). Also, in the SDSS dataset, we did not run the query sequence for any of the MySQL settings, as it took more than 5 hours just to load the data.

Compared to the other methods, VALINOR exhibits significantly lower execution time in almost all cases. Particularly in TAXI dataset, where VALINOR times range between 0.3 to 12 sec, VALINOR is more than 2× faster in all queries and more than 10× faster in 35% of queries than the best competitor, and in the rest of datasets VALINOR is about 2-5× faster.

Regarding PostgresRaw, we observe that it requires approximately the same time for every query. The positional map used in PostgresRaw, attempts to reduce

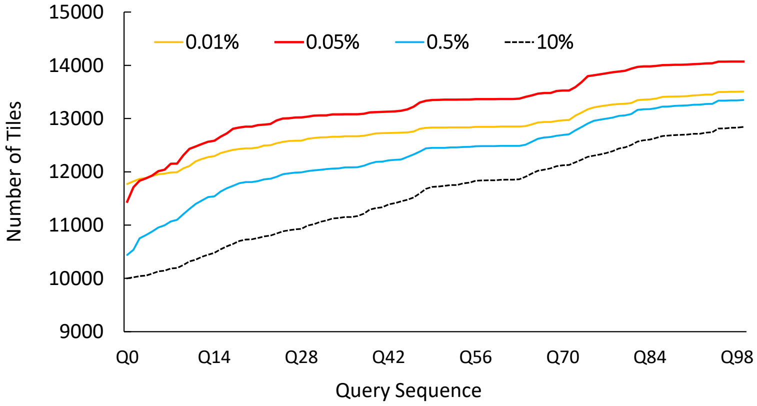
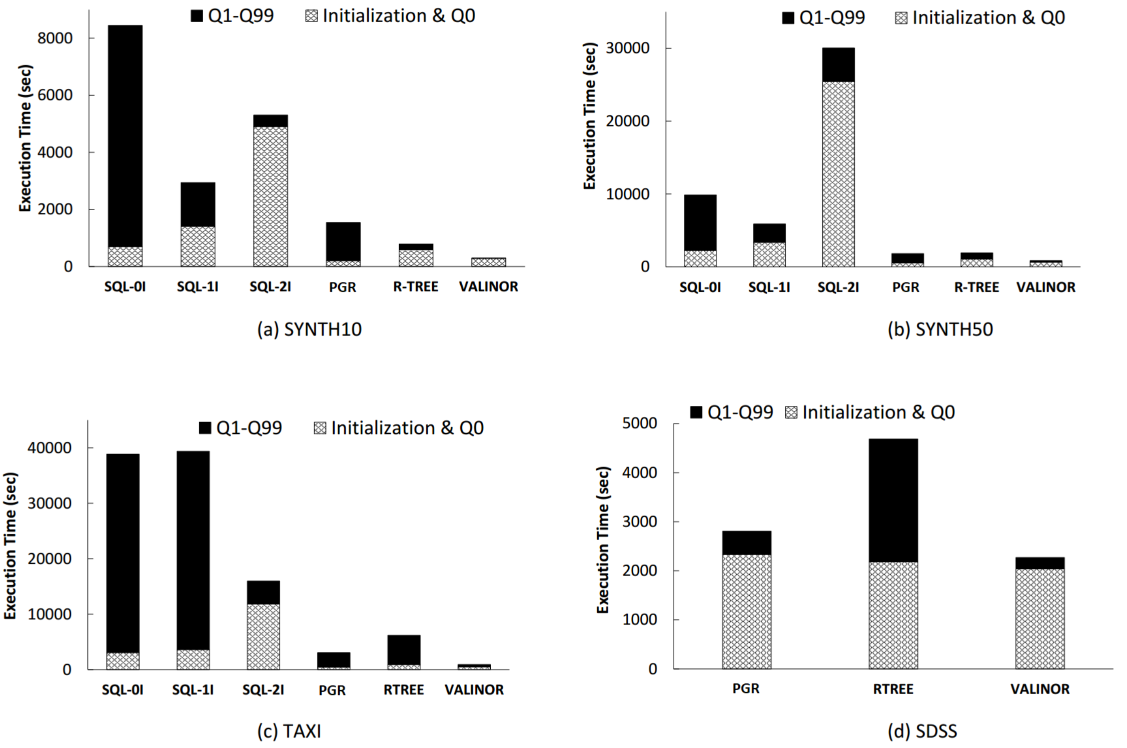


Figure 4.13: Number of VALINOR Tiles varying Q_0 Selectivity

the parsing and tokenizing costs of future queries, by maintaining the position of specific attributes for every object in the raw file. However, PostgresRaw still needs to examine all objects in the dataset in order to select the ones contained in a 2D window query. Also, in contrast to VALINOR, PostgresRaw does not keep any metadata in order to efficiently compute the aggregate queries. This is also the main reason why the R-tree is significantly slower compared to VALINOR. For the evaluation of the analysis clause of a query the R-tree cannot reuse previously computed metadata in order to reduce the number of I/O operations, and has to go to the raw file for every object contained in the selection clause of the query.

Besides the positional map used in PostgresRaw, a cache is also employed to hold the values of previously accessed attributes and avoid access to the raw file altogether. So, for queries $Q_1 \sim Q_{99}$ where the cache is already populated, and the attributes requested are the same as in Q_0, PostgresRaw does not need to access the raw file. The time to execute every query then depends mainly on the number of objects contained in the dataset. For example, for TAXI which contains 165M objects every query takes around 26 sec, while for SDSS which contains 40M objects, 4.7 sec. This explains why PostgresRaw is faster for some of the queries in the SDSS dataset compared to VALINOR. VALINOR, despite adapting to the workload in order to minimize file reads, still needs to access the raw file for the objects of partially-contained tiles. For SDSS, these raw file accesses are particularly expensive considering its disk size (270GB). Nevertheless, VALINOR performs better than PostgresRaw for most of the queries in SDSS, needing approximately 51% less total time to execute queries $Q_1 \sim Q_{99}$.

The accumulative time needed to execute the query sequence of the exploration scenario for every dataset is shown in Figure 4.14. The accumulative time captures the overall performance of the user scenario. This time includes Q_0 which is depicted separately from all subsequent queries. As it can be seen, the cumulative time needed to execute the complete workload by VALINOR is much lower in relation to other systems. For example, for the TAXI dataset VALINOR needs around 15 min, while PostgresRaw, which is the best competitive method for this dataset, requires approximately 51 min. Even though PostgresRaw needs less time to answer the first query for the TAXI dataset, as well as for SYNTH10/50, the rest of the sequence is executed mush faster by VALINOR, resulting in better overall performance.



(a) SYNTH10

(b) SYNTH50

(c) TAXI

(d) SDSS

Figure 4.14: VALINOR Overall Execution Time for the Entire Exploration Scenario

4.6.2.4 Discussion

We observe that VALINOR achieves for most queries (except Q1) of SYNTH10 and SYNTH50 response times between 0.07 and 0.55 and between 0.1 and 0.8, respectively. Note that, in SYNTH10 *only one query reports time more than 0.5 sec*, and in the SYNTH50 dataset 11 queries. Regarding the SDSS and TAXI datasets, due to a noticeable larger number of I/Os (about two orders of magnitude more), the response times are larger. Particularly, in SDSS we have times between 0.15 and 9.5. However, in more than 35% of the queries the time is less than 1 sec. On the other hand, the best competitive method (PostgresRaw), reports times more than 4.2 sec in all queries. In the TAXI dataset, where we have the larger number of I/Os, we have times between 0.3 and 11, with 4.2 seconds being the average value. On the other hand, the best competitive method (PostgresRaw), reports times between 23 and 28, with 26 as average value. Further, in PostgresRaw about 85% of the queries require more than 25 sec. Hence, in 85% of the queries the PostgresRaw reports more than twice worse performance compared to our worst case (11 sec). Overall, in our experiments, the proposed method, in most cases, is about 5-10× faster than the competitors, and requires significantly less memory resources.

Finally, we have to note that the system's performance is highly affected by implementation issues. For example, in our case, the disk I/O operations cost, dominates the response time. The VALINOR has been implemented as a prototype using the Java programming language, which is known to have poor performance in I/O operations, compared to other programming languages; e.g., C/C++. So, the use of other programming languages will have an impact on performance.

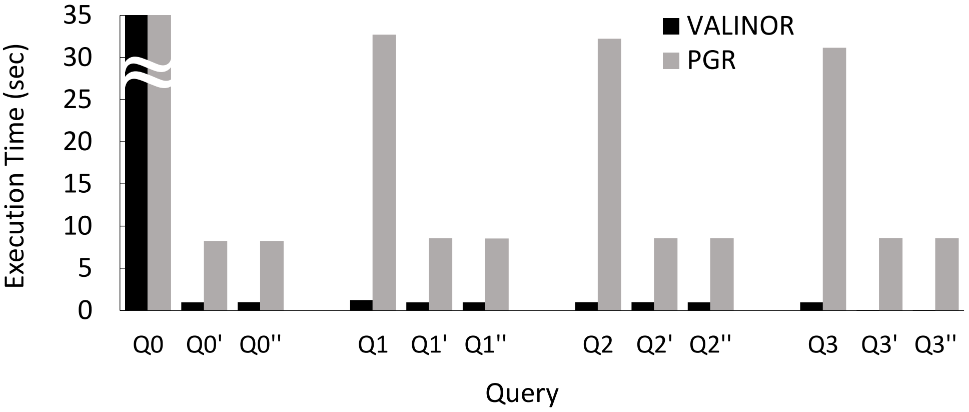


Figure 4.15: VALINOR Execution Time for Filter Operations (SYNTH10)

4.6.2.5 Evaluating Filter Operations

For assessing the behaviour of VALINOR with regard to varying filtering on non-axis attributes, we compare VALINOR against PostgresRaw while varying the filter clause. For this, we generate 4 queries for SYNTH10, keeping the selection clause (i.e. window query) fixed, while alternating their filter condition. Specifically, the filter clause of each query includes a condition over a different non-axis attribute. For example, Q_0 filters objects having their 8th attribute greater than 700; Q_2 filters objects with the 6th attribute less than 200, etc. The same workload of queries Q_0 ~ Q_3 are repeated 3 times and the results are shown in Figure 4.15. In the plot, the first iteration of Q_0 includes the initialization time for both systems, which explains the significantly higher execution time. As can be seen, VALINOR outperforms PostgresRaw in this experiment. For every such query, VALINOR first evaluates the selection clause, and may read the raw file to retrieve the non-axis attribute included in the filter clause only for the objects contained in the window query. Also, while reading these non-axis attributes, it stores tile metadata for them, which assists next filter queries avoid expensive IO operations. This is evident especially for Q_3. The first time Q_3 is executed, there is no tile metadata for the 9th attribute which its filter condition references. As a result, VALINOR needs to retrieve this attribute for all objects contained in the selection clause. Simultaneously, while reading this attribute, it populates fully-contained tiles with related metadata (e.g., min, max for this attribute). When the same query Q_3 is executed again, VALINOR utilizes this metadata to avoid most I/O operations, which explains its faster execution time. Regarding PostgresRaw, we can observe that apart from the first iteration of Q_0, which initializes the positional map and cache of the system and thus exhibits much higher execution time, PostgresRaw also exhibits a significantly slower execution time for the first iteration of Q_1 ~ Q_3 as it populates the positional map for the corresponding non-axis attributes of each query's filter condition. Next iterations of these queries require less time, as they can utilize the already populated structures of PostgresRaw.

4.6.2.6 Combining Pan and Zoom Operations

In this experiment, we compare VALINOR's performance with that of PostgresRaw and R-tree, for the second exploration scenario which includes pan, zoom-in and

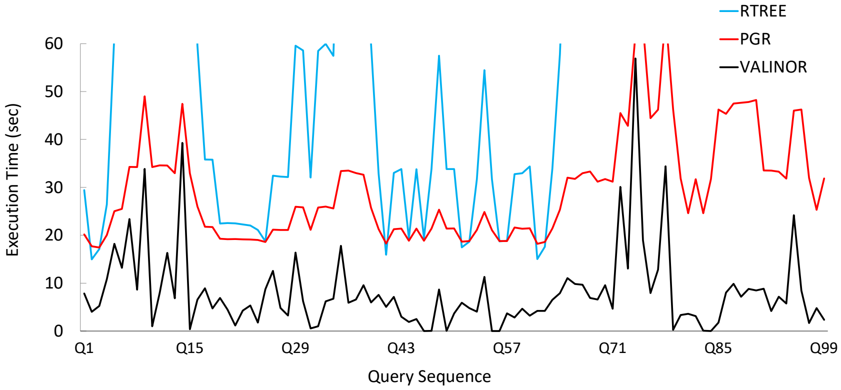


Figure 4.16: VALINOR: Exploration using Pan and Zoom In/Out Operations (TAXI)

zoom-out operations on the TAXI dataset. Figure 4.16 presents the results, where VALINOR exhibits better performance for every query $Q_0 \sim Q_{99}$. Compared to the first exploration scenario which did not include zoom operations, we can observe that in this, query execution times vary significantly. Zoom-out operations increase the number of objects contained in a window query and result in slower execution times in general. For example, Q_9, Q_{13} and Q_{74} correspond to zoom-out operations, which explains the significantly higher execution time observed for all 3 methods examined. On the other hand, zoom-in operations restrict the visualized area and reduce the objects that need to be examined. As a result, queries like Q_2 or Q_{42} correspond to drops in execution time. The aforementioned behavior, where larger window queries result in slower execution time, is more consistent for the PostgresRaw and R-tree methods, where all contained objects have to be examined and their non-axis attributes included in the analysis clause of the query, either fetched from disk for R-tree, or from disk or cache for PostgresRaw. On the contrary, VALINOR performs tile splittings and populates fully-contained tiles with metadata for the non-axis attributes that are retrieved. As a result, even for a zoom-out operation (e.g., Q_{82}), VALINOR may require less time than the previous, smaller window query, if it can utilize tile metadata to avoid I/O operations.

4.6.2.7 Memory Consumption

In this experiment, we examine how VALINOR's size in memory changes while it is adapted to the query workload. The experiment was ran on the synthetic datasets with the index operating using its query-driven initialization and adaptation policies. Note that, the memory consumption in VALINOR is not affected by the objects' dimensionality, since in each case, only the two axis attributes are indexed. As a result, using either of the two synthetic datasets (SYNTH10/50) would require the same memory. The query workload used is the same as in previous experiments, with each query requesting bivariate statistics on two non-axis attributes. Figure 4.17 shows the results. We can observe that the total size of the index increases slightly as queries are processed. This is the result of tile splitting to adapt to the query workload and of metadata being stored for fully-contained tiles.

Figure 4.18 presents VALINOR's memory footprint compared to R-tree. We did not consider PostgresRaw and MySQL settings since they exhibit different memory requirements due to their tight-coupling with the RDBMS. Nevertheless, PostgresRaw required a significant amount of memory for its positional map and

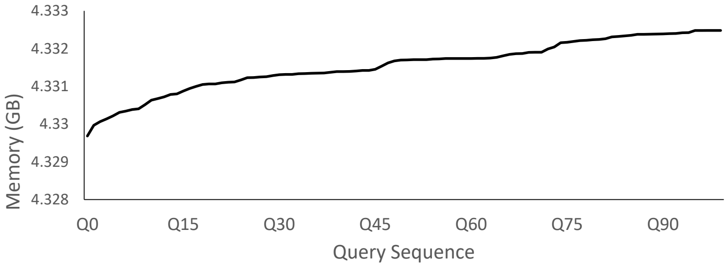


Figure 4.17: VALINOR During an Exploration Scenario

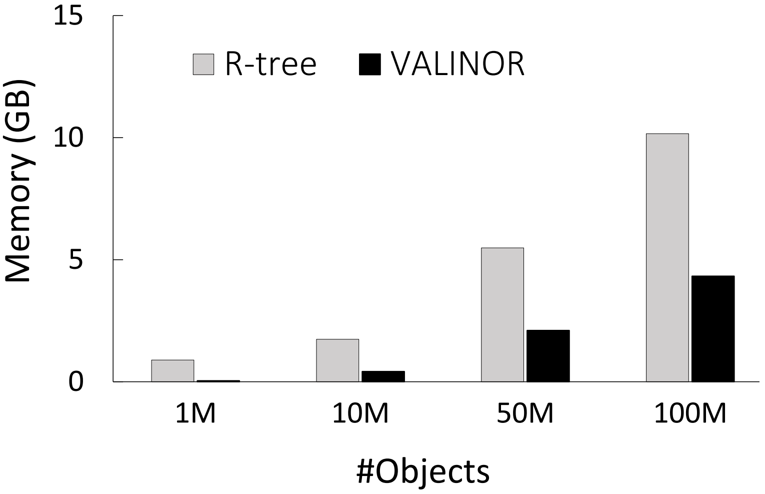


Figure 4.18: VALINOR vs. R-tree

cache for datasets with more attributes (in some cases more than 32GB). For this experiment, we measured the memory used to build VALINOR and R-tree varying the number of objects in the synthetic dataset. Note that, same as VALINOR, the memory of R-tree is not affected by the objects' dimensionality. So, SYNTH10 is the same as SYNTH50. We can observe that VALINOR requires significantly less memory than R-tree, with R-tree requiring 2× more memory for 100M objects.

4.6.2.8 Performance of VALINOR under Memory Constraints.

Next, we examine the behavior of VALINOR when operating under memory limitations and its index structure size exceeds the available memory size. In this scenario, parts of the index have to be evicted to the disk and loaded again into memory as needed. For this experiment, we used the SYNTH10 dataset running the same workload as before, but varying the percentage of objects that can fit into the memory available between 25%, 50%, 75% and 100%. To show the effect of eviction during query processing, we modified the workload used previously for the synthetic dataset, generating sequences of 100 overlapping queries, increasing the window size (5×) and shifting each query in relation to its previous one by a shift amount of 50%.

Figure 4.19 presents the cumulative time needed to answer the query sequence for every case. As we can see, VALINOR's initialization time increases under memory pressure. Since the objects cannot fit in memory, some objects are evicted during the initialization phase. To better demonstrate how memory pressure affects query

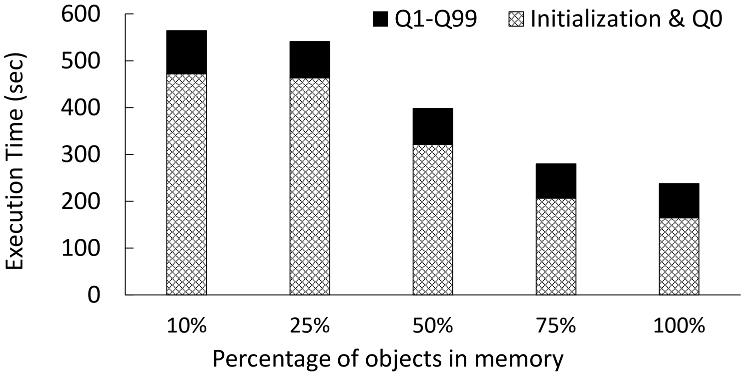


Figure 4.19: Overall Execution Time of VALINOR varying the Memory Size (SYNTH10)

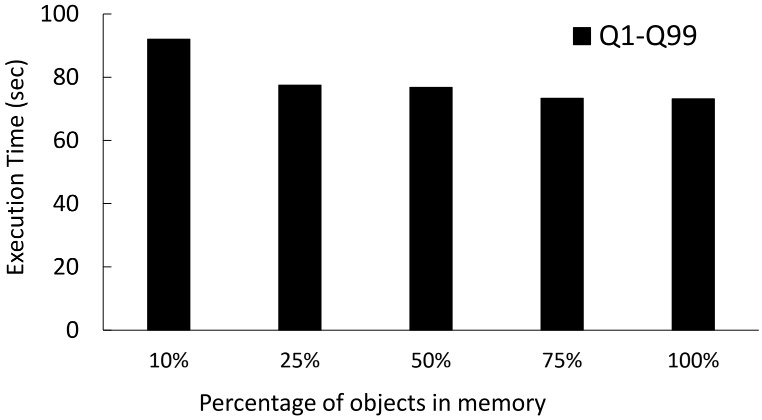


Figure 4.20: Overall Execution Time of VALINOR

processing, we present separately in Figure 4.20 the cumulative time needed to answer Q_1 ~ Q_{99}. As can be seen, the time needed to answer queries after the initial one, is not greatly affected. Since we follow an eviction policy based on the distance from the query, and the workload consists of neighboring and overlapping queries, very few evictions need to happen during query processing. Specifically, the effect is more pronounced when restricting the available memory to 10% of the dataset, as can also be seen in Figure 4.21, which presents the execution time for queries Q_1 ~ Q_{99}.

4.7 Summary

This chapter has introduced the VALINOR index, a core component of our proposed approach for efficient visual exploration and analytics over large raw data files. Our focus has been on the scenarios where users interact with data in a 2D visual exploration context, and the index is optimized to handle such scenarios, particularly when the data is based on two numeric attributes.

We outlined the hierarchical, tile-based structure of the VALINOR index and discussed its role in grouping objects for efficient access and analysis. The aggregated metadata associated with each tile not only enriches the index but also speeds up analytic operations.

We delved into the user-driven initialization algorithm for building the index.

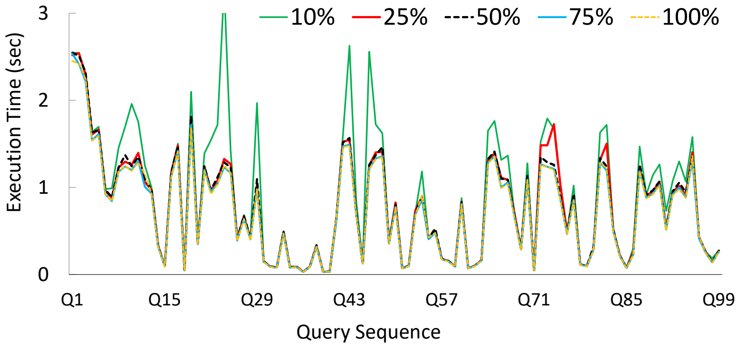


Figure 4.21: Execution Time of VALINOR varying the Memory Size (SYNTH10)

The novelty of this approach lies in its usage of a locality-based probabilistic approach, facilitating faster user interaction at the initial stages of the exploration.

The chapter further explained our unique query-based adaptation technique. This method incrementally adjusts the index structure based on user interaction, enhancing performance, particularly for analytic tasks.

We also touched upon the practical considerations of implementing such a system, discussing the index's space complexity and our mechanism for dealing with memory constraints. The eviction mechanism provides a balanced approach to memory management, allowing parts of the index to be stored on disk when necessary.

The chapter concluded with an extensive experimental evaluation of the proposed methods and their performance. The results demonstrated the superiority of our approach compared to competitive systems in terms of execution time, I/O operations, memory consumption, and scalability.

In conclusion, the VALINOR index is a robust solution for enhancing the speed and efficiency of 2D visual exploration and analytic operations over large raw data files. As we progress into the next chapter, our emphasis will shift towards the adept handling of visual analysis over categorical attributes, with a specific focus on the efficient execution of *group* and *filter* operations on these attributes.

Chapter 5

Indexing for Visual Exploration over Categorical Attributes

5.1 Introduction

Visual exploration and analysis of raw data is a challenging yet critical task in many domains. This task often involves multiple visual techniques that leverage various types of attributes, such as numerical and categorical. In the previous chapter, we introduced the VALINOR index, which mainly focuses on 2D exploration scenarios involving two numeric attributes. While this approach is highly effective for visual scenarios like scatter plots or geospatial maps, it is not directly applicable to exploratory visual analysis that also uses categorical attributes. Many common visual techniques, such as bar charts or heat maps, rely heavily on categorical attributes. The efficient handling of such attributes in in-situ exploration scenarios presents significant challenges, which we attempt to address in this chapter.

Considering the visual exploration model presented in Chapter 3, VALINOR was designed to efficiently evaluate operations such as *render*, *move*, *zoom in/out*, as well as the *analyze* operation which calculates aggregate statistics over the entire 2D window query. However, in this chapter, our focus shifts towards operations involving categorical attributes, such as *filter*, *group*, and the *analyze* operations to compute aggregate statistics per group. Group-by analysis is required to generate well-known visualization types, such as bar charts, heatmaps, parallel coordinates, binned scatter plots, radar charts, pies, etc. The importance of categorical-based visualization types in data analysis is verified by [68], showing that bar charts are the most commonly used visualization type. Many of these charts and interactions are largely employed in common data analysis tasks, such as feature extraction, OLAP analysis, regression, and comparative analysis of spatial data [68]. Beyond visual analytics requiring group-by operations, filter operations over categorical attributes enable support for effective exploration mechanisms, such as faceted search. These types of analysis and queries have been widely optimized in traditional data warehouse systems, via spatial and multidimensional indexes, and with pre-aggregated materialized views over the data. However, these methods require loading the data, tuning the indexes and materializing the aggregated views.

The in-situ visual exploration scenario that this work focuses on attempts to avoid the overhead of moving, loading, and indexing the data in a DBMS and to improve performance by progressively adapting an index as the user explores data.

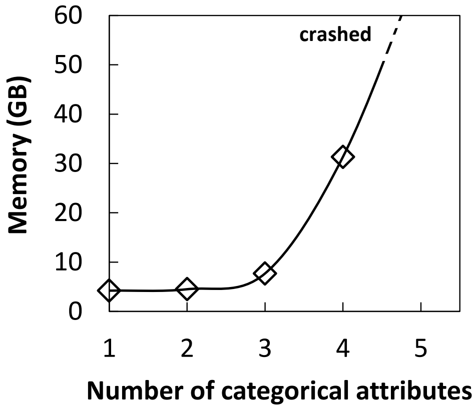


Figure 5.1: Indexing Memory Requirements vs.Number of Categorical Attributes

The key objective is to offer fast user interactions without a preprocessing phase. Despite the challenges faced when performing 2D in-situ visual exploratory analysis that the previous chapter attempted to address (e.g., on-the-fly index construction, small data-to-analysis time, reducing the cost of I/O operations to the raw data files), in cases where categorical attributes are involved, the memory required to index the dataset becomes prohibitive even for a small number of attributes.

For example, Figure 5.1 shows the memory allocated for our initial "crude" version of VALINOR expanded to also index the objects based on the categorical attributes, over different numbers of attributes, on a dataset with 100M objects (SYNTH10 dataset, Sect.5.7). We observe that for 4 categorical attributes, the size of the index is 31GB, while indexing 5 attributes requires more than 64GB of memory. These amounts of memory are not usually available in commodity hardware-based scenarios. The challenge here is: what part of the data do we choose to index and how do we optimize the index given a predefined memory size?

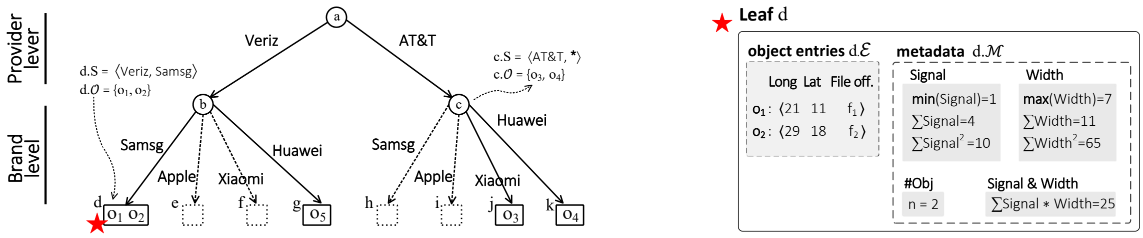
Moreover, another challenge is related to the efficient query evaluation over the index in exploratory operations involving categorical attributes. The metadata stored in the tiles of VALINOR refer to all the objects of the tile, and cannot be utilized if a query involves a *Group-by clause* or even a *Filter clause* that involves a categorical attribute. Thus, the challenge here is: how to effectively enrich the index so that it can efficiently evaluate such categorical-based operations?

To address these challenges, in this chapter, we propose an indexing scheme called the *VETI index*, designed to cater specifically to in-situ visual analytics scenarios that involve categorical-based operations, including group-by and filter operations, in combination with 2D visual interactions and statistics. This index extends VALINOR, and is built on top of the tile-based structure, which efficiently supports visual exploration over the 2D plane. Additionally, it is enhanced with a tree-based structure that organizes a tile's objects based on their categorical values. This dual structure offers an effective approach to handling both categorical and numerical attributes, thereby making it highly adaptable to a range of visual exploration and analytic scenarios.

This chapter will cover the key aspects of the *VETI index*, including the *CET tree* for enhancing the VALINOR tile structure with categorical-based indexing, the query evaluation and index adaptation mechanism, the resource-aware index initialization approach, and the proposed algorithms for handling the corresponding optimization problem. To demonstrate the effectiveness and robustness of our

Table 5.1: VETI Notation

Symbol	Description
$\mathcal{O}$, o_i	Set of objects, an object
$\mathcal{A}$, $a_{i,A}$	Set of attributes, the value of attribute A of the object o_i
A_x, A_y, $\mathcal{A}_C$	X Y Axis & Categorical attributes
$\mathcal{C}$	Ordered set of categorical attributes
Q, $\mathcal{R}$	Exploratory Query, its Results
$\mathbb{I}$, $\mathbb{I}_{\mathcal{T}}$	VETI index; its Tiles
h, $h.\mathcal{C}$	CET tree; its Categorical attributes
$h.N$	Number of CET nodes
$t.h$	Tree h of tile t
ρ_t, ρ_h	Tile & Tree utility
$HP_{\mathcal{C}}$	Attributes-based Tree Powerset, given a set $\mathcal{C}$
π_t^h, $\pi_t^h.\omega$	A Tile-Tree Assignment and its Utility
$\mathbb{I}_\Pi$, $\Omega(\mathbb{I}_\Pi)$	Index Assignments; Index Utility
$\mathcal{B}$	Initialization memory budget
$\pi_t^h.\Phi$	Memory cost estimation for assignment π_t^h
$\mathcal{H}$	Candidate tree set
$\mathbb{I}_{cost}$, $\mathbb{I}_{\mathcal{T}cost}$, $\mathbb{I}_{\mathcal{H}cost}$	Memory cost of: index $\mathbb{I}$, its tiles $\mathbb{I}_{\mathcal{T}}$ and its trees $\mathbb{I}_{\mathcal{H}}$



(a) **CET Tree** (b) **Contents of Leaf** d

Figure 5.2: CET Tree Overview

approach, we will provide a detailed performance evaluation based on real and synthetic datasets. Our results highlight the remarkable speed and efficiency of our methods, achieving interactive query response times even over large raw files.

5.2 CET Tree: An Index for Categorical Attributes

5.2.1 Design Principles

In this section, we present a *tree structure that organizes objects based on their categorical attribute* values, named CET (Categorical Exploration Tree). CET is designed as a *lightweight*, *memory-oriented*, *trie-like* tree structure. In a nutshell, each tree level corresponds to a different categorical attribute, and edges to attribute values. Based on the tree hierarchy, each node is associated with a set of objects, that are determined based on the node path. These objects are stored in the leaf nodes.

Overall, the design of the CET tree relies on the following principles and challenges. First, considering the number of attribute-value combinations which are required for categorical indexing, a significant amount of memory is required (Fig. 5.1). Hence, the design of a *memory-efficient categorical structure* is a major challenge, especially in our scenario, where we consider limited available resources. To reduce the memory footprint of the tree, we implement the following techniques: (1) *Each object allocates three numeric values*: (a) two numeric values for the axis attributes; and (b) one numeric value (i.e., file offset) that offers object-based, precise "connection" between object and raw file (2) *Statistics are stored only in one tree level* (in leaves), while the hierarchical structure of CET allows the efficient computation of statistics over different levels, by performing efficient, in-memory aggregate operations.

(3) *The number of tree elements is reduced* (i.e., nodes/edges) during tree construction, by considering attribute characteristics, i.e., size of the attributes' domain (see Sect. 5.2.3).

A second challenge is to *reduce the cost of* I/O *operations* which are crucial in such I/O-sensitive settings. Exploiting the way CET stores the objects during the initialization phase (Sect. 5.3.2), we are able to *access the raw file in a sequential manner.* As we previously noted in the context of the VALINOR index, performing a sequential file scan enhances the number of I/O operations over contiguous disk blocks, consequently improving the utilization of the look-ahead disk cache. This is confirmed by our experiments, in which *the sequential access results in about* 8× *faster* I/O *operations* (more details in Sect. 5.4.1).

5.2.2 CET Structure

In this section, we present the basic concepts of the CET tree. Given a set of objects $\mathcal{O}$ and an ordered set (list) of categorical attributes $\mathcal{C} = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$, a CET tree h organizes the objects $h.\mathcal{O}$ based on the values of the categorical attributes $h.\mathcal{C}$. The *height* of h is $|\mathcal{C}|$, so it has $|\mathcal{C}| + 1$ *levels* (from 0 to $|\mathcal{C}|$), with the *leaf nodes* storing the objects.

CET follows a "level-based" organization, where *each level corresponds to a different attribute.* Specifically, based on the given order of the attributes $\mathcal{C}$, the *nodes at level i have edges* that correspond to a different value of the attribute $A_{C_i} \in \mathcal{C}$, i.e., $dom(A_{C_i})$.

Each node n, is *associated with a sequence of attribute values $n.S = \langle v_0, v_1 ..., v_k \rangle$*, that is defined by the *path from the root to node n.* The sequence contains $|\mathcal{C}|$ values, where the value v_i corresponds to a value of the attribute in level i. Specifically, for a node n at the level i, the first i^{th} values in $n.S$ are the attributes values found in the path from the root to n, while the rest $|\mathcal{C}| - i$ values are assigned with the value *any*, denoted as $*$.

Based on the sequence of values $n.S$, a node *is associated with a set of objects $n.\mathcal{O} \in \mathcal{O}$*, where its attribute values are equal to the sequence's values. As a result, the tree defines an aggregation structure, where in each node, the associated objects are the union of the objects associated with its child nodes. Note that, to reduce the memory requirements of the index, we maintain a hash table for each categorical attribute mapping its values to numeric hashes.

Object Entries. *Leaf nodes* contain references to the data objects, i.e., object entries. Note that, object entries are not included in internal nodes. As in the case of VALINOR, for each object $o_i \in n.\mathcal{O}$, an *object entry e_i* is defined as $\langle a_{i,x}, a_{i,y}, f_i \rangle$, where $a_{i,x}, a_{i,y}$ are the values of the axis attributes and f_i the offset (a hex value) of o_i in the raw file. As $n.\mathcal{E}$ we denote the set of object entries stored in the leaf node n. In any case, an object entry has a constant size that is not affected by the object's characteristics (e.g, number of attributes), and is equal to three numeric values: the object's A_x and A_y (e.g., two double), and the object's offset from the beginning of the file, e.g., a long. The file offset f_i defines a "direct and precise" object-based connection between an object and the raw file.

Synopsis Metadata. Apart from object entries, each leaf node n is associated with a set of *synopsis metadata* $n.\mathcal{M}$, which are (numeric) values calculated by algebraic aggregate functions [38] over one or more attributes of the $n.\mathcal{E}$ objects.

Recall that, combining the algebraic aggregate functions allows us to support a large number of statistics, e.g., Pearson correlation, covariance.[2] For example, we employ functions like *sum*, *mean*, *sum of squares of deltas* over the objects of a leaf.

Using leaf metadata, we are able to compute the metadata of any internal node n, by aggregating the metadata of the descendant nodes of n, in a bottom-up fashion.

Example 6. [**CET Tree**] Figure 5.2a presents the CET index constructed for the categorical attributes $\mathcal{C} = \{A_{Provider},\ A_{Brand}\}$ of a raw data file with network signal measurements, a sample of which is presented in 3.1a. The dotted lines indicate parts of the tree that will not be constructed for the particular dataset.

Considering the level-based organization, the *level* 0 corresponds to the *Provider* attribute (the first attribute in $\mathcal{C}$), and *level* 1 to *Brand.* The nodes in each level have as *edges* the values of the level's corresponding attribute, e.g., edges of node a are the *Provider* values: *Provider* = {Ver, AT&T}.

Also, the node c has the *associated sequence values* $c.S$ = ⟨AT&T, $\ast$⟩, where AT&T corresponds to the path of c, and the value *any* is produced by the absence of the *Brand* attribute (in the path). Further, c is *associated with the objects* $c.\mathcal{O} = \{o_3, o_4\}$ that "match" with the $c.S$ values, i.e., have as *Provider* the value *AT&T* and the value *any* for *Brand.*

Regarding the *leaf nodes*, the leaf d stores the object entries $d.\mathcal{E}$ and the metadata $d.\mathcal{M}$ for the objects $d.\mathcal{O} = \{o_1, o_2\}$ that matches its values $d.\mathcal{S}$ = ⟨Veriz, Samsg⟩ (Fig. 5.2b). Here, metadata stores statistics regarding the *Signal* and the *Width* numeric attributes.

5.2.3 CET Operations

This section presents the basic operations of the CET tree and analyzes their computational complexity.

5.2.3.1 Insert & Tree Construction

Insertion takes as input, a tree h, an object o, and an ordered set of categorical attributes $\mathcal{C} = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$ and inserts o in a leaf node based on the values $A_{\mathcal{C}_i}$ of its categorical attributes, constructing new edges and nodes for the values that do not exist in the tree. Also, the leaf metadata is updated w.r.t. the o numeric attributes. The tree *construction* is implemented via sequential *insert operations* of its objects.

The *computation complexity* of the *insert* operation is $O(|\mathcal{C}|)$, and that of *construction* considering n objects is $O(n\,|\mathcal{C}|)$.

5.2.3.2 Get Leaves/Objects Based on Filter Conditions

The *get leaves operation* returns the leaf nodes $\mathcal{L}$ of a tree h. Based on the conditions in the Filter clause F of a query, the operation constructs paths p starting from the root to the leaf nodes and returns the leaves $\mathcal{L}$ reached by all paths. The *get objects operation* returns the object entries of the leaves $\mathcal{L}$.

Regarding the *computation complexity* of the *get leaves* and *get objects* operations, the worst case occurs when we have to access all the leaf nodes in the tree. In that case, the complexity is $O(h.N)$ and $O(h.N + |h.\mathcal{O}|)$ respectively, where $h.N$ is the number of nodes in the tree and $h.\mathcal{O}$ its objects.

5.2.3.3 Expand Tree with New Attributes

The *expand tree* operation adds new levels in the tree and reorganizes the objects in the leaves. It is used when a query requests attributes not existing in the tree. In such cases, the values of the missing attributes retrieved from the file expand the tree (see Sect. 5.4). The operation takes as input the new categorical attributes $\mathcal{C}$, and a subset of leaf nodes $\mathcal{L}$ of a tree h, which should be reorganized based on $\mathcal{C}$. For each leaf node $l_i \in \mathcal{L}$, a subtree h_i having l_i as root is constructed, where h_i has one level for each attribute $A_C \in \mathcal{C}$ and leaf nodes $\mathcal{L}_{h_i}$. The objects of each leaf node l_i are organized based on $\mathcal{C}$ attributes and stored to the leaf nodes $\mathcal{L}_{h_i}$ of the generated tree h_i. Further, the metadata of the new leaf nodes $\mathcal{L}_{h_i}$ are computed.

Note that after the *expand tree* operation, the leaf nodes of the tree may appear at different levels, as only the subset of leaf nodes needed to evaluate a query are expanded with the new attributes. This way, we avoid unnecessary I/O operations by reading only the attributes for the objects included in the query. Otherwise, we would need to read the new attributes for every object in the tree in order to fully create the new attribute levels.

Regarding, the *computational complexity*, the worst case appears when the leafs $\mathcal{L}$ to expand, enclose all the tree objects $h.\mathcal{O}$. In such a case, the complexity is $O(|h.\mathcal{O}|\,|\mathcal{C}|)$.

5.2.4 Tree Space Analysis

Considering the CET insertion process, nodes are created based on the values of the objects being inserted in the tree. We can easily verify that the maximum number of nodes in a CET tree occurs when all possible combinations of values for its attributes appear in the objects it contains. Given the tree attributes $h.\mathcal{C} = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$, the *maximum number of nodes* $h.N$ is: $1 + |\mathrm{dom}(A_{C_0})| + |\mathrm{dom}(A_{C_0})| \cdot |\mathrm{dom}(A_{C_1})| + ... + |\mathrm{dom}(A_{C_0})| \cdot |\mathrm{dom}(A_{C_1})| \cdot ... \cdot |\mathrm{dom}(A_{C_k})| = 1 + \sum_{i=0}^{|h.\mathcal{C}|-1} \prod_{j=0}^{i} |dom(A_{C_j})|$. Note that the term "1" corresponds to the root node.

Considering that a leaf node is created only if it is associated with at least one object, the maximum number of leaf nodes is equal to the number of objects. Similarly, at each level of the tree the number of nodes cannot be larger than the number of objects. In what follows, we consider the *number of objects, in order to define a tighter upper bound for the total number of nodes.*

The maximum number of nodes can be determined using the following recursive formula: $\Gamma_0 = 1$ and $\Gamma_i = min(\Gamma_{i-1} \cdot |\mathrm{dom}(A_{C_{i-1}})|, |h.\mathcal{O}|)$, with $1 \le i \le |h.\mathcal{C}|$. So, if we consider the number of objects is much greater than the product of the size of the attribute domains, we have that the *maximum number of nodes is*: $1+\sum_{i=1}^{|h.\mathcal{C}|}\Gamma_i$.

Since the memory for each node is almost the same (except for the leaves where metadata is stored), here, for simplicity, we assume that all nodes allocate equal memory. Furthermore, all object entries have the same size (about four numeric values). Therefore, the *space complexity* of CET is: $O(\alpha+\alpha\sum_{i=1}^{|h.\mathcal{C}|}\Gamma_i+\beta\,|h.\mathcal{O}|)$, where α and β are the memory allocated by a node and an object entry, respectively.

5.2.5 Attributes Ordering vs. Tree Space

Based on the complexity analysis, we can easily verify that the number of nodes in a tree h depends on the mapping of its attributes $h.\mathcal{C}$ to the levels of the tree and the size of their domain. Assuming that the data follow a uniform distribution over the domain values of each attribute, we can reduce the number of nodes (and edges) in the tree, by placing the attributes at the levels of the tree in a top-down way based on their domain size, i.e., smaller domains are placed closer to the root. So, *constructing a tree following this attribute order, will result in lower space requirements.* In our experiments, this attribute order led to up to 10% reduction in total index memory requirements, compared to a random order (Fig. 5.14).

5.3 VETI: A Tile-Tree Adaptive Index

In this section, we present the VETI indexing scheme (Visual Exploration Tile-Tree Index), that combines the tile-based index presented in Ch. 4 and the CET tree structure to support efficient exploratory operations over the 2D space along with analytics operations based on the values of the categorical attributes. The design of VETI relies on the basic challenges posed by the in-situ exploration scenarios. First, the *index construction should entail a small overhead in the raw data-to-analysis time.* To this end, a lightweight, "crude" version of VETI is initially constructed on-the-fly, by parsing the raw file once. Moreover, the characteristics of this initial VETI version are defined by considering query and data-related factors in order to improve the performance of the initial user interactions. Second, during the exploration, the *index should support efficient exploratory and analytic operations.* Thus, based on user exploration, efficient structure adaptation and object reorganization are employed to adjust the index to user interactions. Third, considering the *limited available resources*, VETI uses lightweight tree and tile structures with predefined memory resources allocated to them (Sect. 5.5).

5.3.1 VETI: Combining Tiles and Trees

VETI is built on top of the VALINOR tile-based index, expanding its *tile-structure* to further support the efficient evaluation of operations that involve the categorical attributes. To this end, VETI combines the tile-structure of VALINOR with the CET trees presented in Section 5.2.

VETI is defined as follows: given a *raw data file* $\mathcal{F}$, two *axis attributes* A_x and A_y, and a set $\mathcal{C}$ of *categorical attributes* of the objects stored in $\mathcal{F}$, the *VETI index*

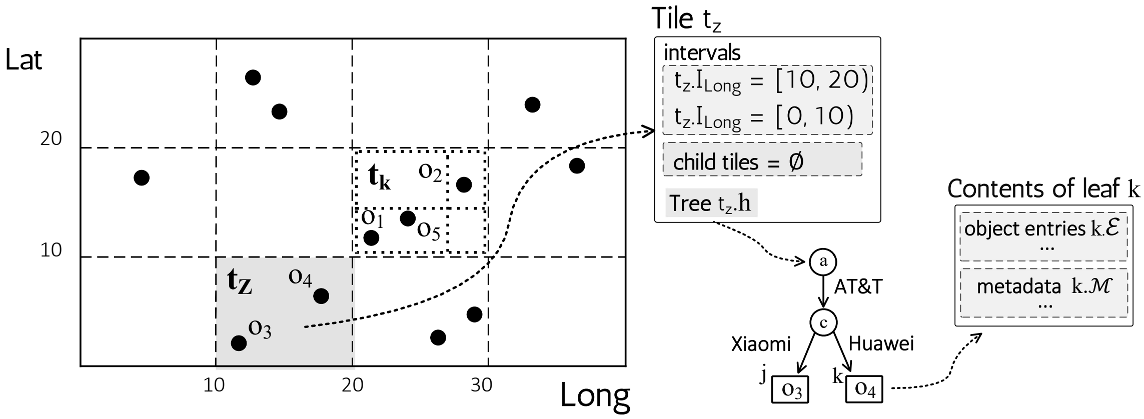


Figure 5.3: VETI Index Overview

$\mathbb{I}$ organizes the objects stored in $\mathcal{F}$ into hierarchies of non-overlapping tiles based on its A_x, A_y values, where tiles are also associated with CET trees which organize objects based on categorical attributes from $\mathcal{C}$.

Let $\mathbb{I}_{\mathcal{T}}$ be the tiles of $\mathbb{I}$. Each leaf tile $t \in \mathbb{I}_{\mathcal{T}}$ *is associated with a CET tree* h, denoted as $t.h$. The associated tree $t.h$ of a tile t, organizes the objects $t.\mathcal{O}$ enclosed by t, based on a set of categorical attributes $\mathcal{C}' \subset \mathcal{C}$, i.e., $h.\mathcal{O} = t.\mathcal{O}$ and $h.\mathcal{C} = \mathcal{C}'$. Thus, trees of different tiles may organize their objects based on different sets of categorical attributes.

The objects $t.\mathcal{O}$ enclosed in a tile t are stored in the leaf nodes of the associated tree $t.h$ and can be accessed via a pointer to the root node of the tree $t.h$. In case the objects of a tile are not indexed based on any categorical attribute (i.e., $h.\mathcal{C} = \varnothing$), the tree h corresponds to a node (root) that stores all the object entries.

The VETI *index* $\mathbb{I}$ is defined by a tuple $\langle \mathbb{I}_{\mathcal{T}}, \mathsf{IT}, \mathsf{IH}, \mathsf{AS} \rangle$, where $\mathbb{I}_{\mathcal{T}}$ is the tile structure (along with the its trees) defined in the index; IT is the *tiles initialization strategy* defining the methods that determine the characteristics of the tile structure; IH is the *tree initialization strategy* defining the methods that determine the characteristics of the tree structures over the tiles; AS is the *adaptation strategy* defining the methods for reconstructing the tiles and trees based on user interaction.

The basic operations of VETI are: *initialization* (Sect. 5.3.2), *query evaluation* (Sect. 5.4), and *adaptation* (Sect. 5.4.2).

Example 7. [**VETI Index**] Figure 5.3 presents the contents of tile t_z, highlighted with grey color in the index, that contains objects o_3 and o_4. For tile t_z, the index stores its intervals $t_z.I_{Lat}$ and $t_z.I_{Long}$, its child tiles $t_z.\mathcal{C}$, and a pointer to its tree $t_z.h$, which contains nodes a, c, j, and k. Finally, the contents of the leaf node k are shown in the figure (we omit presenting the detailed object entry and the metadata).

5.3.2 VETI Initialization

In our in situ visual exploration approach, we do not require any preprocessing or loading phase. VETI is constructed on-the-fly when the user first requests to visualize the file. During the initialization phase, the following tasks are realized. First, the characteristics of the index are determined, i.e., the initial set of tiles and the structure of each tree assigned to each tile are defined; then, the file is parsed

```
Algorithm 3. VETI Initialization ($\mathcal{F}$, $A_x$, $A_y$, $\mathcal{C}$, $Q_0$, $\mathcal{B}$)
  Input: $\mathcal{F}$: raw data file;   $A_x$, $A_y$: axis attributes;
         $\mathcal{C}$: categorical attributes;   $Q_0$: first query
  Output: $\mathbb{I}$: initialized index;   $\mathcal{R}_0$: result of query $Q_0$
1 $\mathbb{I}_\mathcal{T} \leftarrow$ IT.constructTiles($A_x, A_y, \mathcal{A}_\mathcal{C}, Q_0$)   //determine the number, size & intervals of the tiles, and construct them
2 $\mathbb{I}_\Pi \leftarrow$ find tile-tree assignments   //see Sect. 5.5
3 foreach $o_i \in \mathcal{F}$ do   //read objects from file, insert them to trees & evaluate $Q_0$
4     read from $\mathcal{F}$ the values of axis and categorical $\mathcal{C}$, and the attributes required to
      evaluate the $Q_0$ Analysis clause
5     use the $o_i$ attributes to evaluate $Q_0$
6     $t_i \leftarrow$ find the tile $t_i \in \mathbb{I}_\mathcal{T}$ that encloses $o_i$ based on its axis attributes values
7     insertToTree($t_i.h, \mathcal{C}, o_i$)   //insert $o_i$ to tree $t_i.h$ (Proc. 1, Sect. 5.2.3)
8 return $\mathbb{I}$, $\mathcal{R}_0$
```

and the index is populated; finally, the first user query is evaluated.

Algorithm 3 outlines the initialization phase. It takes as input, the raw file $\mathcal{F}$, the axis and categorical attributes A_x, A_y and $\mathcal{A}_\mathcal{C}$, and the first exploratory query Q_0; and returns the initialized index $\mathbb{I}$ and the results $\mathcal{R}_0$ of the Q_0.

Initially, the tile structure characteristics are determined (i.e., number, size and intervals of the tiles) and the tiles $\mathbb{I}_\mathcal{T}$ are constructed (*line* 1). Next, based on the constructed tile structure, the assignments $\mathbb{I}_\Pi$ of trees to tiles $\mathbb{I}_\mathcal{T}$ are determined (*line* 2). Details about the assignment selection methods are presented in Section 5.5.

In the next part (*loop in line* 3), the algorithm scans the file $\mathcal{F}$ and reads, for each object o_i, the values of the axis attributes $a_{i,x}$, $a_{i,y}$, the categorical attributes, and the attributes which are required to evaluate the Analysis clause of Q_0 (*line* 4). Next, the tile t_i that encloses o_i is found (*line* 6), and the insertToTree method (Procedure 1), inserts o_i into the tree $t_i.h$ (*line* 7). During the insertion, the object entry is constructed, the tree metadata are updated, and new parts (i.e., nodes, edges) of the tree may be constructed.

Tile Structure Initialization. The constructTiles method is defined by the tile initialization strategy IT, and determines the tile structure characteristics, e.g., number, size and intervals of the tiles. As already discussed in 4.3.1 and 4.4.1, these characteristics can be defined via numerous approaches (for more details see [79]). For instance, they can be either given explicitly by the user, e.g., in a map the user defines a default scale of coordinates for the initial visualization; determined by the visualization setting, considering certain characteristics (e.g., visualization type, screen size/resolution), previous sessions, task, user preferences [79, 52, 12, 89]; or computed based on techniques that consider data characteristics in order to divide the data space into tiles of equal size, like the baseline initialization approach employed by VALINOR.

For the initialization of VETI, we use the *query-driven initialization policy* for initializing the tiles discussed in 4.4.1. Recall that this policy considers: (1) the user exploration entry point, i.e., the position of the first user query in the 2D space; (2) the window size of the first query; and (3) the locality-based behavior of the exploration scenarios, i.e., users are more likely to explore nearby regions of their initial entry point [102, 54, 89, 12, 96, 26]. In a nutshell, the query-driven tile initialization defines a tile structure that is more fine-grained (i.e., has a larger

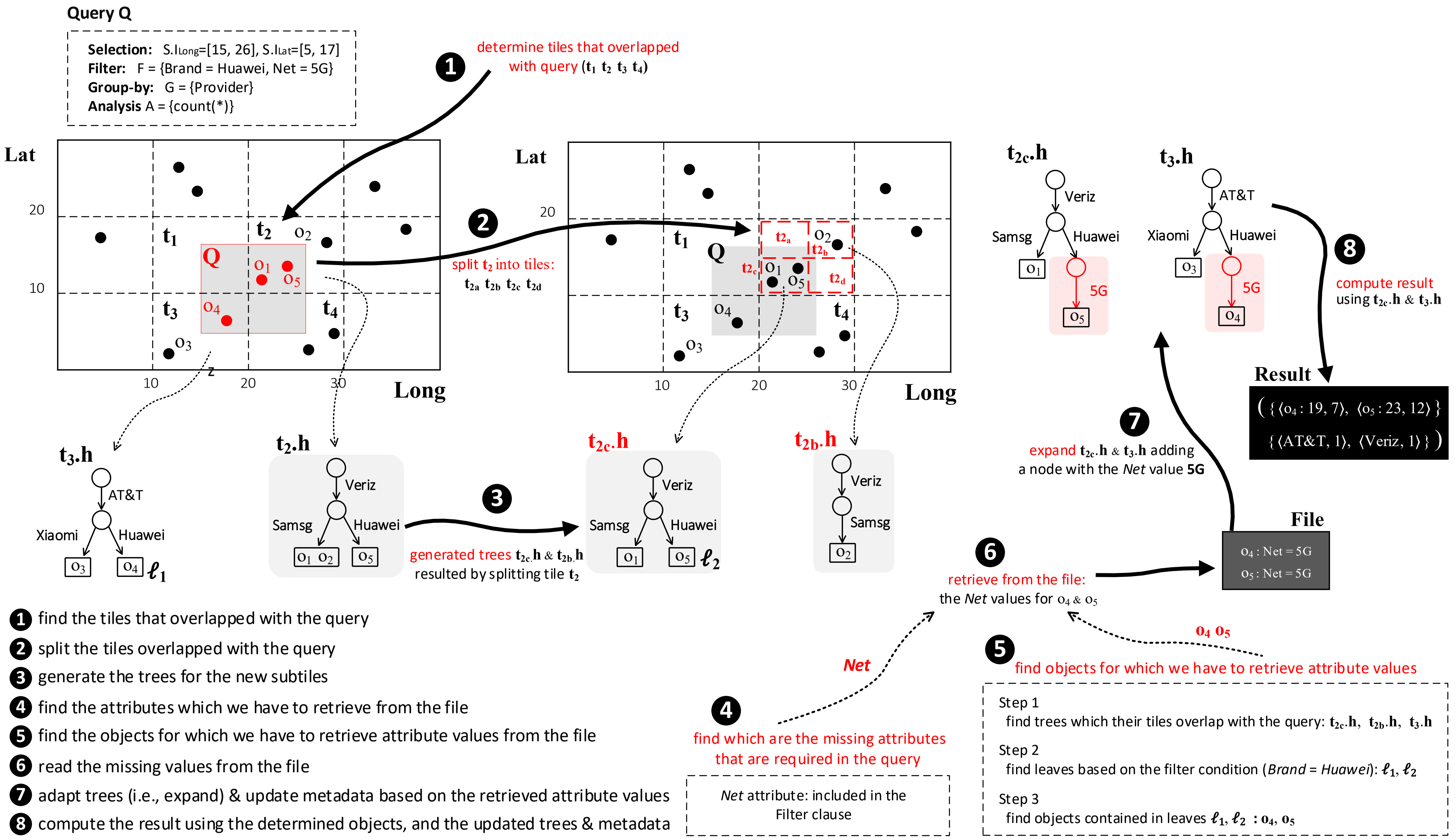

Query Q
Selection: S.I$_{Long}$=[15, 26], S.I$_{Lat}$=[5, 17]
Filter: F = {Brand = Huawei, Net = 5G}
Group-by: G = {Provider}
Analysis A = {count(*)}
1 determine tiles that overlapped with query (t_1 t_2 t_3 t_4)
Lat
Long
10
20
30
t_1
t_2
t_3
t_4
Q
o_1
o_2
o_3
o_4
o_5
2 split t_2 into tiles: t_{2a} t_{2b} t_{2c} t_{2d}
t_{2a}
t_{2b}
t_{2c}
t_{2d}
t_3.h
AT&T
Xiaomi
Huawei
ℓ_1
t_2.h
Veriz
Samsg
Huawei
3 generated trees t_{2c}.h & t_{2b}.h resulted by splitting tile t_2
t_{2c}.h
ℓ_2
t_{2b}.h
5G
8 compute result using t_{2c}.h & t_3.h
Result
({ ⟨o_4 : 19, 7⟩, ⟨o_5 : 23, 12⟩ } { ⟨AT&T, 1⟩, ⟨Veriz, 1⟩ })
7 expand t_{2c}.h & t_3.h adding a node with the Net value 5G
File
o_4 : Net = 5G
o_5 : Net = 5G
6 retrieve from the file: the Net values for o_4 & o_5
Net
o_4 o_5
4 find which are the missing attributes that are required in the query
Net attribute: included in the Filter clause
5 find objects for which we have to retrieve attribute values
Step 1
find trees which their tiles overlap with the query: t_{2c}.h, t_{2b}.h, t_3.h
Step 2
find leaves based on the filter condition (Brand = Huawei): ℓ_1, ℓ_2
Step 3
find objects contained in leaves ℓ_1, ℓ_2 : o_4, o_5
1 find the tiles that overlapped with the query
2 split the tiles overlapped with the query
3 generate the trees for the new subtiles
4 find the attributes which we have to retrieve from the file
5 find the objects for which we have to retrieve attribute values from the file
6 read the missing values from the file
7 adapt trees (i.e., expand) & update metadata based on the retrieved attribute values
8 compute the result using the determined objects, and the updated trees & metadata

Figure 5.4: VETI Query Processing & Index Adaptation Example

number of smaller tiles) in the area around the initial query, whereas the size of tiles becomes larger as their distance from the initial query increase.

Increasing the number (i.e., decreasing the size) of tiles near the first query, increases the probability that subsequent queries in this neighborhood overlap with fully-contained tiles, which in turn reduce the number of I/O's. More details about query evaluation are presented in the next section.

Discussion. Note that, beyond CET trees, we also studied alternative structures for indexing categorical attributes in VETI. Specifically, we considered the use of bitmap structures which are effective for indexing low cardinality attributes and are highly compressible. In brief, in a VETI variation we implemented, we combined the tile structure with a set of bitmap structures instead of CET trees. In this variation, based on the objects $t.\mathcal{O}$ of a tile t, a bitmap structure is constructed for each distinct value of every categorical attribute in $t.C$.

As in our approach, apart from the object entries, each bitmap is also associated with a set of metadata pertaining to its objects indexed in it.

In our experiments, as it was expected, the bitmap variations requires, in general, less memory than the VETI with CET trees. This is not only the result of the use of the highly compressible bitmaps, but also because the bitmap variations maintain metadata for single categorical attribute values and not for different combinations. On the other hand, the query evaluation performance is significantly lower. The limited metadata stored in the bitmap variations cannot be utilized to avoid I/Os for queries that involve two or more categorical attributes. As a result, VETI is in most queries more than 2-3× faster and requires less than the half I/O operations compared to the bitmap implementations.

5.4 Query Processing & Index Adaptation

This section describes the query processing methods of VETI. Figure 5.4 outlines the workflow, whose steps are described in the following example[1]. More details on the query evaluation over the index are given in Section 5.4.1 and on index adaptation in Section 5.4.2.

Example 8. [***Query Processing & Index Adaptation***] As input we have the initialized index, an exploratory query and a raw file. Considering the objects in Figure 1.1, we assume an exploratory query Q with the following clauses (left upper corner in Fig. 5.4) : (1) *Selection clause*: S with $\mathsf{S}.I_{Long}$=[15, 26] & $\mathsf{S}.I_{Lat}$=[5, 17]; (2) *Filter clause*: $\mathsf{F} = \{Brand$ = Huawei, Net = 5G$\}$; and (3) *Group-by clause*: $\mathsf{G} = \{Provider\}$; and (4) *Analysis clause*: $\mathsf{L} = \{count(*)\}$.

Further, we assume that the index is initialized and every tile has a tree with the attributes *Provider* and *Brand*. Additionally, the tree leaves contain aggregate metadata for the *Signal* attribute. The query processing and index adaptation are depicted in Figure 5.4.

❶ To evaluate query Q we first *find the leaf tiles that spatially overlap* (i.e., partially or fully-contained) with its Selection clause, i.e., t_1, t_2, t_3, t_4. ❷ Next, we

[1] Note that, since several details are omitted, the order of the steps may be different compared to the following paragraphs, where the process is presented in detail. Also, in the implementation, several of these steps are performed in parallel.

check if the overlapping tiles need to be split, in such case, the tiles are split into smaller subtiles. The tile splitting may be performed based on different methods, such as: equally or arbitrary-sized splitting. In each splitting step, the process considers criteria related to I/O cost in order to decide whether to perform a split or not (more details at Sect. 5.4.2). In our example, we assume that t_2 is split into four equal disjoint subtiles: t_{2_a}, t_{2_b}, t_{2_c}, t_{2_d}. ❸ Then, the objects are reassigned to the new subtiles and their *trees are generated*; here, the trees $t_{2_c.h}$ and $t_{2_b.h}$ of the new subtiles t_{2_b} and t_{2_c}.

❹ We, then, find *the attributes of the query which are not contained in the index*, and for which their values have to be retrieved from the file. In our example, the query's Filter clause includes conditions over the *Brand* and *Net* attributes, i.e., *Brand* = Huawei, *Net* = 5G. Also, the Group-by clause contains the *Provider* attribute. Since the index was initialized to include the categorical attributes *Brand* and *Provider*, values for the *Net* attribute are not available in the index.

❺ We *determine the objects for which we have to read the NET attributes from the file.* For that, considering the tiles that overlapped with the query (t_{2_a}, t_{2_b}, t_{2_c}, t_{2_d}, t_3, t_4), we *identify the trees of these tiles* ($t_{2_b}.h$, $t_{2_c}.h$ and $t_3.h.$) and we traverse each tree for identifying their *leaves which evaluate to the query's condition Brand* = Huawei. These leaves are ℓ_1 and ℓ_2 from the trees $t_{2_c}.h$ and $t_3.h$, which contain the objects o_4 and o_5. ❻ For these objects, we *read the file* and retrieve the *Net* attribute values.

❼ Based on the values retrieved from the file, *trees are adapted/expanded* (e.g., create new nodes/edges, reorganize leaf objects) in order to include the new attribute and *update the metadata.* Here, using the retrieved *Net* values of o_4 and o_5, the trees $t_{2_c}.h$ and $t_3.h$ are *expanded* to include the new categorical attribute *NET* (see **expand** operation, Sect. 5.2.3).

❽ Finally, query Q *is evaluated* on the objects o_4 and o_5 for the condition *Net* = 5G, and on the tree metadata for the *Group by* and *Analysis* clauses. In our example, the *count* function is calculated by the number of objects in each leaf node. The result consists of: the tuples of the selected objects o_4 and o_5 and their axis attributes $(\{\langle o_4 : 19, 7\rangle, \langle o_5 : 23, 12\rangle\})$; and the tuples that form the result of the *Group by* and *Analysis* clause $(\{\langle AT\&T, 1\rangle, \langle Veriz, 1\rangle\})$.

5.4.1 Query Processing

The main query processing is presented in Algorithm 4. The algorithm also includes the index adaptation phases, which are analyzed in the next section. The algorithm takes as input, the initialized index $\mathbb{I}$, an exploratory query Q and the raw file $\mathcal{F}$. Next we provide details on the implementation of each step presented in the previous example.

Find and Adapt Query's Overlapped Tiles & Trees. Once the index has been initialized, the algorithm finds the leaf tiles $\mathcal{T}_{\mathsf{S}}$ that overlap with the 2D area defined in the query's Selection clause S (*line* 2). The function getLeafTiles OverlappedWithQuery determines the overlapping tiles at the highest-level, and then traverses the tile hierarchy to find the set of overlapping leaf tiles $\mathcal{T}_{\mathsf{S}}$.

Next, based on the adaptation strategy AS, the adaptTileAndTree procedure (*line* 4), performs the tile splitting and reorganizes the trees (constructing new or modifying

Algorithm 4. VETI Query Processing ($\mathbb{I}$, Q, $\mathcal{F}$)

```
Input: 𝕀: index (initialized);  Q⟨S, F, D, G, L⟩: query;  ℱ: raw data file
Variables: $\mathcal{T}_S$: leaf tiles that overlap with the Selection clause, i.e., 2D area; $\mathcal{T}_a$: tiles resulted from
           adaptation;  $\mathcal{L}$: tree leaf nodes selected by the Query;  $\mathcal{W}(\langle l,\mathcal{V}\rangle)$: set of tuples $\langle l,\mathcal{V}\rangle$,
           where $\mathcal{V}$ are objects' attributes, and $l$ its leaf
Parameters: AS: adaptation strategy;
Output: $\mathcal{R}$: result of query $Q$

1  $\mathcal{L} \leftarrow \varnothing$
2  $\mathcal{T}_S \leftarrow$ getLeafTilesOverlappedWithQuery ($\mathbb{I}_{\mathcal{T}}$, S)
3  foreach $t_s \in \mathcal{T}_S$ do
4      $\mathcal{T}_a \leftarrow$ AS.adaptTileAndTree ($t_s, Q$)                         //see Sect. 5.4.2
5      $\forall t_a \in \mathcal{T}_a$ :  $\mathcal{L} \leftarrow \mathcal{L} \cup$ getLeavesBasedOnFilter ($t_a.h$, F)     //Procedure 2 (Sect. 5.2.3)

6  $\mathcal{W}(\langle l,\mathcal{V}\rangle) \leftarrow$ getLeavesRequiringFileAccess ($\mathcal{L}, Q$)  //set of tuples, where $\mathcal{V}$ are the attributes of a leaf $l$ whose their values need to
   be retrieved from the file

7  if $\mathcal{W} \neq \varnothing$ then                                            //values are missing — read from file
8      read from file the values of attributes $\mathcal{V}$ for the objects of leaf $l$,  $\forall\langle l,\mathcal{V}\rangle \in \mathcal{W}$

9      expandTree ($l$, $\mathcal{V}$)  $\forall\langle l,\mathcal{V}\rangle \in \mathcal{W}$   //update tree based on retrieved attributes; i.e., expand tree's leaf $l$ with its missing attributes $\mathcal{V}$
       (Sect. 5.2.3)

10     updateLeafMetadata ( $l$ )  $\forall l \in \mathcal{W}$

11 $\mathcal{R} \leftarrow$ evaluate $Q$ using the objects and the metadata of leaves $\mathcal{L}$
12 return $\mathcal{R}$
```

existing) that are included in the tiles $\mathcal{T}_a$ created by the splitting process (more details in Sect. 5.4.2). Finally, considering any conditions over categorical attributes that are defined in the Filter clause, getLeavesBasedOnFilter retrieves the leaf nodes $\mathcal{L}$ of the $\mathcal{T}_a$ trees (*line* 5).

Determine the Objects that Require File Access. After identifying the tiles overlapping with the query and the corresponding leaves, we determine the objects for which we have to access the raw file in order to answer the query.

Procedure getLeavesRequiringFileAccess $(\mathcal{L}, Q)$ (*line* 6), first, considers the spatial relation between the 2D area specified in a Select clause and the tiles it overlaps. Specifically, a *tile t* that overlaps a query Q can be *partially-contained* or *fully-contained* in Q. So, the procedure for each leaf node in $\mathcal{L}$, first checks if the tile it belongs to, is partially or fully-contained in the query Q. In the case that a leaf belongs to a *partially-contained tile*, the leaf metadata can not be used, since only a subset of a (leaf's) objects could be selected by the query. Hence, we need to find the objects of the leaf that are contained in the query; then, for these objects, we retrieve from the file the attributes required to compute the metadata and evaluate the Analysis clause of the query.

Apparently, in the case that a leaf belongs to a *fully-contained tile*, we do not need to traverse its objects in order to find the ones that are included in the window and the tile's metadata can be used without the need to access the file. In fully-contained tiles, file access is needed only when the query refers to attributes for which *information is not stored in the index*, e.g., *Net* attribute in the query example.

Based on the aforementioned, the procedure getLeavesRequiringFileAccess identifies the attributes, whose values have to be retrieved from the file. Finally, it returns a list $\mathcal{W}$ of tuples $\langle l, \mathcal{V}\rangle$, where $\mathcal{V}$ are the attributes that must be retrieved for the objects included in the leaf l.

Read Objects' Attributes from File. To *reduce the cost of reading the missing attributes from file (line 8)*, we exploit the way the object entries are stored in the leaves in order to access the file in a sequential manner. During the initialization of the index, we append the object entries into the leaf nodes of the CET trees as the file is parsed. As a result, object entries in every leaf node are stored sorted based on their file offset. When accessing the file, we read the objects from the leaves following a *k-way merge* based on their file offset. Thus, we are able to *access the raw file in a sequential manner.* The sequential file scan increases the number of I/O operation over continuous disk blocks and improves the utilization of the look-ahead disk cache. Note that, in our experiments, *the sequential access results in about* 8× *faster* I/O *operations* compared to accessing the file by reading objects on a "leaf basis", i.e., read the objects of leaf l_i, then read the objects of tile l_k, etc.

Adapt Trees and Update Metadata based on the Attributes Read from File. Next, based on the attributes for which values are read from the file, the trees (of fully-contained tiles) are adapted/enriched to include the retrieved attributes. Particularly, the expandTree procedure (Sect. 5.2.3) adapts/enriches the trees by including the retrieved attributes and reorganizes the objects (*line* 9). As already discussed in Section 5.2.3, the expandTree procedure expands the trees only for the objects that it needs to read from the file to evaluate the query. This partial tree expansion adapts the trees with new attributes without performing unnecessary I/O operations. Then, the function updateLeafMetadata computes and updates the metadata using the values retrieved from the file (*line* 10).

Evaluate Query. Finally, we evaluate query Q using the objects and metadata of the leaf nodes $\mathcal{L}$ (*line* 11). Here we use the attribute values retrieved from the file to check the filter conditions that do not involve categorical attributes. Also, we need to check the objects belonging to trees missing some of the categorical attributes included in the Filter clause. Finally, the Group-by and Analysis clauses are evaluated using: (1) existing metadata of the fully-contained tiles, if their corresponding CET trees include all the categorical attributes of the query; (2) for all other cases, the values retrieved from the file.

5.4.2 Incremental Index Adaptation

VETI employs an *incremental index adaptation* model that attempts to adapt the index structure to the query workflow of the user exploration. Each query may result in splitting the tiles overlapping the Selection clause into smaller subtiles. *Tile splitting increases the likelihood that a tile included in the area that the user exploration focuses on, will be fully-contained in a future query* and the use of metadata in fully-contained tiles will reduce the number of file accesses, improving the query performance. For that reason, splitting is performed as a first step of the query evaluation process, such that we compute metadata for the new subtiles and then evaluate the query over a more fine-grained index. Specifically, it is performed after function getLeafTiles OverlappedWithQuery has determined the leaf tiles that overlap with the Selection clause (*line* 2).

Procedure adaptTileAndTree (*line* 4) is responsible for the incremental adaptation. It takes as input a tile t and a query Q and returns a set of subtiles $\mathcal{T}_a$ if t needs to be split. To determine if a tile t requires (further) splitting, we estimate the *expected splitting gain* in terms of I/O cost, for evaluating a (future) query Q, in case of splitting t. If the expected splitting gain for a tile, exceeds a given splitting threshold, a split is performed. A more detailed analysis of the splitting model was presented in 4.4.3.

In our implementation for VETI, the I/O cost is formulated by the selectivity of Q over t, where selectivity is computed by the number of objects in t and the filter conditions defined in Q.

Tile Splitting. After the tile splitting, the adaptTileAndTree (*line* 4) procedure returns a set of subtiles $\mathcal{T}_a$. Each one of the children contains a tree with the same set of categorical attributes as their parent tile. The objects contained in the leaf nodes of the parent tile's tree are reorganized in the leaf nodes of the new trees according to their values for the axis attributes, as well as the categorical attributes.

As with VALINOR, we can employ various approaches for splitting a tile. For example, we can employ a quad-tree-like splitting approach in which a tile is split into 4 equal subtiles or the more sophisticated query-based splitting method introduced in 4.4.2.

Reorganize Trees in Splitted Tiles. As discussed in Section 5.2.3, the order of the attributes in a tree affects its size (number of nodes/edges). Hence, during splitting, the attributes of the trees that are generated in the new subtiles, are sorted so that the attributes with smaller domain sizes are placed closer to the root. For this, we consider the distinct values of the categorical attributes within the bounds of the parent tile t. Then, we reorganize the objects of t into the trees of the children $\mathcal{T}_a$.

To reorganize the objects, we perform Depth-first search in the tree $t.h$ to iterate over all of its leaf nodes. Based on the path of every leaf node from the root, we can determine the values of its categorical attributes. Then, for each object entry of a leaf node, we find the subtile that encloses it and we insert it into its tree (using the *insert* operation).

Adaptation Computation Complexity. The overall computational cost of tile splitting, consists of the cost of splitting the tile t, constructing $\mathcal{T}_a$, and reorganizing the objects $t.\mathcal{O}$ in $\mathcal{T}_a$ trees. First, we have to determine the intervals of $\mathcal{T}_a$, and define the subtiles as child tiles of t, i.e., initialize the child pointers. These can be performed in constant time $O(1)$. Then, we perform Depth-first search (DFS) in the tile's tree h, and reinsert its objects into the trees of the subtiles. The cost of DFS is $O(h.N)$, where $h.N$ is the number of nodes in the tree, and the cost of the *insert* operation is $O(|h.\mathcal{C}|)$. So, the overall cost is $O(h.N + |t.\mathcal{O}|\,|h.\mathcal{C}|)$.

5.5 Resource-aware Index Initialization

In this section, we present the initialization of the CET trees and their assignment to tiles. Recall from Figure 5.1, that more than 64GB is required for VETI to create full trees from five categorical attributes. Our goal here is to determine the structures of the trees (the categorical attributes that will be placed as levels in the tree) and assign them to tiles based on the "utility" of each tree. The latter depends on the utility of the categorical attributes it contains; we consider that an attribute has a higher utility score when its inclusion in the tile's tree is expected to improve *the performance in the user exploration scenario.*

We define the *ReSource-aware INdex Initialization* (SIN) problem and formulate and solve it as an optimization problem of assigning trees to tiles based on the utility score. In what follows, we first provide some preliminaries and then define the SIN problem.

5.5.1 Preliminaries

Before we formally introduce our problem, we present some necessary definitions. [2]

Tile Utility. Let a tile t, the *tile utility* $\rho_t \in [0,1]$ formulates the possibility that a future exploratory query will overlap with t. For the distribution of ρ_t, we follow the approach presented in 4.4.2. Based on the locality-based characteristics of 2D exploration scenarios, users are more likely to explore nearby regions of their initial exploration entry point [102, 54, 89, 12, 96, 26]. Thus, given an initial query Q_0, the next queries are more likely to overlap with tiles near Q_0 and the value of ρ_t is larger in tiles near Q_0. Particularly, as we presented in 5.4.2 the probability that a query overlaps with a tile t is modeled considering a bivariate normal distribution based on the distance of the tiles from the center of Q_0.

Attribute Score & Tree Utility. We assume that *each categorical attribute c has a score $c.S \in [0,1]$*, that represents the probability that a future query will request this attribute. We define the attribute score based on the "repetitive calculation of statistics" that appears in exploration scenarios [94], i.e., we assume that the attributes requested by the initial query Q_0, are more likely to be requested by next user interactions. Using the attributes scores, the tree utility is defined as follows.

Given a tree h, the *tree utility* $\rho_h \in [0,1]$ formulates the possibility that an exploratory query requests information stored in the tree. Without loss of generality, we define the *tree utility* ρ_h as the normalized sum of the scores of the tree attributes $h.\mathcal{C}$:

$$\rho_h = \frac{\sum\limits_{\forall c \in h.\mathcal{C}} c.S}{\sum\limits_{\forall c \in \mathcal{C}} c.S} \tag{5.1}$$

Example 9. [***Running Example***] Consider a VETI index with six *tiles* (t_1- t_6); three categorical attributes *Provider* (P), *Brand* (B) and *Net* (N), with domain size 2, 4, and 3, respectively; and a *query* Q_0 that includes a *Group-by clause* on attribute P, and a *Filter clause* on attribute B. We assume that Q_0 overlaps with t_1 and based on the other tiles' position, the *tile utilities* are: $\rho_{t_1} = 0.6$, $\rho_{t_2} = 0.1$, $\rho_{t_3} = 0.1$, $\rho_{t_4} = 0.1$, $\rho_{t_5} = 0.05$, and $\rho_{t_6} = 0.05$.

Regarding the *categorical attribute scores*, the attributes P and B are included in Q_0 and assigned with a score 0.8, whereas N has score 0.1. Additionally, assume the trees: $h_{P,B}.\mathcal{C} = \{P, B\}$, and $h_{P,N}.\mathcal{C} = \{P, N\}$. The tree $h_{P,B}$ that includes both

[2] Note that, several of the problem's involved metrics (e.g., tile and tree utility) can be computed using a large number of factors, e.g., device and visualization type, interface, user profile, task, domain [79]. However, this is beyond the scope of this work.

attributes of Q_0 will have a larger utility than $h_{P,N}$ which includes only one of them. Based on the Eq. 5.1 the *tree utilities* are $\rho_{h_{P,B}} = 0.96$ and $\rho_{h_{P,N}} = 0.82$.

Tile-Tree Assignment. A *tile-tree assignment* (or simply *assignment*) π_t^h, assigns *a tree h to a tile t.* So, given a tile t and a tree x an assignment π_t^x defines that $t_i.h = x$.

Tile-Tree Assignment Utility Each *tile-tree assignment π_t^h is associated with a utility* $\pi_t^h.\omega \in [0,1]$, which formulates the possibility that a query is going to request information from the tile t involving the attributes $h.\mathcal{C}$ of its tree. Intuitively, the utility formulates the "effectiveness of the information" contained by a tile-tree assignment during query evaluation. The *tile-tree assignment utility* is defined as the joint probability of the *tile utility* ρ_t and the *tree utility* ρ_h:

$$\pi_t^h.\omega = \rho_t \cdot \rho_h \tag{5.2}$$

Attributes-based Tree Powerset. Given a set of categorical attributes $\mathcal{C}$, the *attributes-based tree powerset* $HP_{\mathcal{C}}$, contains the trees generated by considering all possible subsets of $\mathcal{C}$. That is $2^{|\mathcal{C}|}$ trees, containing also the tree with no attributes, i.e., empty tree.

Index Assignments. Given a VETI index $\mathbb{I}$, its tiles $\mathbb{I}_{\mathcal{T}}$, and the categorical attributes $\mathcal{C}$; the *index assignment set* $\mathbb{I}_{\Pi}$ contains all the tile-tree assignments defined in the index tiles $\mathbb{I}_{\mathcal{T}}$, i.e., $\mathbb{I}_{\Pi} = \{\pi_t^h : t \in \mathbb{I}_{\mathcal{T}} \text{ and } h \in HP_{\mathcal{C}}\}$.

Example 10. [***Assignments***] Consider the index of the Example 9. The *attributes-based tree powerset* for the attributes P, B, N is: $HP_{\{P,B,N\}} = \{h_{P,B,N}, h_{P,B}, h_{P,N}, h_{B,N}, h_P, h_B, h_N, h_{emp}$
An assignment over $\mathbb{I}$ can include any tree from this set, e.g., the *index assignment set* $\mathbb{I}_{\Pi} = \{\pi_{t_1}^{h_{P,B,N}}, \pi_{t_2}^{h_N}, \pi_{t_3}^{h_{P,B,N}}\}$

assigns the tree $h_{P,B,N}$ to tiles t_1, t_3; h_N to t_2, and no assignments (i.e., empty tree) are made for tiles t_4, t_5, t_6.

Index Utility. The *index utility* Ω of the entire index $\mathbb{I}$ is the sum of the utilities of all tile-tree assignments $\mathbb{I}_{\Pi}$ made in the index, which is defined as:

$$\Omega(\mathbb{I}_{\Pi}) = \sum_{\forall \pi_t^h \in \mathbb{I}_{\Pi}} \pi_t^h.\omega \tag{5.3}$$

Index Initialization Cost. The *index initialization cost* $\mathbb{I}_{cost}$ denotes the resources (e.g., memory, time) that are required for the VETI initialization. Here, as *resource* we only refer to memory. Specifically, the index initialization cost denotes the memory allocated by the index structures (i.e., tiles, trees, metadata), and does not include the memory required by the object entries that allocate a constant amount of memory; each object allocates three numeric values (Sect. 5.2).

This cost includes: (1) the $\mathbb{I}_{\mathcal{T}cost}$ of constructing the tiles $\mathbb{I}_{\mathcal{T}}$, which is mainly the memory allocated for the tile intervals, pointers to subtiles, and the pointers connecting tiles and trees; and (2) the $\mathbb{I}_{\mathcal{H}cost}$ of constructing the CET trees of the tiles (i.e., the trees defined in the tile-tree assignments), which is the memory

Table 5.2: SIN Example: Tile-tree Assignment Utilities

	Tree						
Tile	$h_{P,B,N}$ (33)	$h_{P,B}$ (9)	$h_{P,N}$ (11)	$h_{B,N}$ (16)	h_P (2)	h_B (4)	h_N (3)
$\boldsymbol{t_1}$ (0.6)	0.60	0.56	0.32	0.32	0.28	0.28	0.04
$\boldsymbol{t_2}$ (0.1)	0.10	0.09	0.05	0.05	0.05	0.05	0.01
$\boldsymbol{t_3}$ (0.1)	0.10	0.09	0.05	0.05	0.05	0.05	0.01
$\boldsymbol{t_4}$ (0.1)	0.10	0.09	0.05	0.05	0.05	0.05	0.01
$\boldsymbol{t_5}$ (0.05)	0.05	0.05	0.03	0.03	0.02	0.02	0.00
$\boldsymbol{t_6}$ (0.05)	0.05	0.05	0.03	0.03	0.02	0.02	0.00

allocated for the tree nodes, edges and metadata stored in the leaf nodes. Thus, the VETI *initialization cost* is: $\mathbb{I}_{cost} = \mathbb{I}_{\mathcal{T}cost} + \mathbb{I}_{\mathcal{H}cost}$.

Index Initialization Budget. We assume an *index initialization budget* $\mathcal{B}$, which is the upper bound of the index initialization cost $\mathbb{I}_{cost}$. In other words, $\mathcal{B}$ denotes the maximum memory size that can be allocated during the initialization.

5.5.2 Problem Definition & Analysis

The *Re<u>S</u>ource-aware <u>IN</u>dex Initialization* problem is defined as follows.

Problem 1. [Resource-aware Index Initialization Problem (SIN).] Given a set of objects $\mathcal{O}$ with categorical attributes $\mathcal{C}$, a set of tiles $\mathbb{I}_{\mathcal{T}}$, and a budget $\mathcal{B}$; our goal is to find the index tile-tree assignments set $\mathbb{I}^{*}_{\Pi}$ of a VETI index $\mathbb{I}$ with tiles $\mathbb{I}_{\mathcal{T}}$, such that the index utility Ω is maximized and the index initialization cost $\mathbb{I}_{cost}$ is lower than the budget $\mathcal{B}$.

$$\Omega(\mathbb{I}^{*}_{\Pi}) = \arg\max \Omega(\mathbb{I}_{\Pi}) \quad \text{and} \quad \mathbb{I}_{cost} \leq \mathcal{B}$$

Example 11. [***SIN Problem***] Based on Example 9, we assume the six tiles ($t_1 - t_6$) and the attributes P, B, N. Table 5.2 presents the tile-tree assignment utilities (Eq. 5.2) for all the possible assignments over the tiles, the tiles utilities (in parenthesis), and the cost of every possible tree (in parenthesis). Here, the cost of the trees is expressed in number of tree nodes, and we assume that all combinations of attribute values appear in the data (for space complexity see Sect. 5.2.3). For example, based on the domain of the attributes P, B, N (Example 9) the tree $h_{P,B,N}$ has cost (number of nodes) equal to 33. Also, the assignment $\pi_{t_1}^{h_{P,B,N}}$ that assigns tree $h_{P,B,N}$ to tile t_1 has utility $\pi_{t_1}^{h_{P,B,N}}.\omega = 0.6$.

In order to solve SIN from Table 5.2 we have to determine the tile-tree assignments that maximize the total index utility and keeps the assignment cost lower than the available budget. Let 50 be the budget available for the tree structures, expressed in total number of tree nodes in the index. We can verify, that the index assignment set $\mathbb{I}_{\Pi} = \{\pi_{t_1}^{h_{P,B}}, \pi_{t_2}^{h_{P,B}}, \pi_{t_3}^{h_{P,B}}, \pi_{t_4}^{h_{P,B}}, \pi_{t_5}^{h_{P,B}}, \pi_{t_6}^{h_P}\}$ corresponds to a solution of SIN. Particularly, these assignments result in a total index utility $\Omega(\mathbb{I}_{\Pi})$ equal to 0.9 (which is the largest), and the cost $\mathbb{I}_{\mathcal{H}cost}$ of its trees is 47.

Theorem 1. The SIN problem is NP-hard.

Proof Sketch. We reduce our problem to the 0-1 Knapsack Problem (KP), which is known to be NP-hard and which states that there is a bin with a capacity, and a set of items. Each item has a weight and a profit. The goal is to find a set of items that maximizes the sum of the profits and the sum of weights is lower than the bin's capacity.

We consider a *restricted instance* of SIN, where: (1) the index contains one tile; (2) the tile utility is equal to one;

(3) each attribute has a construction cost (i.e., the memory overhead when it is included in a tree); and the tree cost is the sum of its attributes' costs.

We reduce SIN to KP via the following *associations*: (1) bin to tile; (2) bin capacity to memory budget minus the cost for constructing the tiles; (3) item to categorical attribute; (4) item profit to attribute score; and (5) item weight to attribute construction cost. We can verify that, the index utility in SIN corresponds to the total profit in KP; and the budget constraint to the capacity constraint, respectively. ■

5.6 SIN Algorithms

In this section, we propose two approximation algorithms in order to solve the SIN problem.

The optimal solution of the SIN problem would be to examine the utility scores of all possible tree assignments from the powerset $HP_{\mathcal{C}}$ to the tiles $\mathbb{I}_{\mathcal{T}}$, and select the set of assignments that maximizes the total utility and its index initialization cost is lower than the memory budget. In the worst case we have to examine $O(2^{|\mathcal{C}||\mathbb{I}_{\mathcal{T}}|})$ tile-tree assignments (including empty trees).

In what follows we present two approximation algorithms to solve the SIN problem. The algorithms is based on two concepts: they examine a subset of *candidate trees* from the powerset $HP_{\mathcal{C}}$, in order to prune the space of the possible assignments; and they estimate a memory cost for the trees in order to handle the budget constraint. In what follows, we define the basic concepts.

5.6.1 Preliminaries

Candidate Trees. The *candidate trees* is a subset of the $HP_{\mathcal{C}}$ set, that contains $|\mathcal{C}|$ trees with "promising" categorical attributes, i.e., the ones that are expected to increase the index utility. To determine the promising attributes, we sort the attributes $\mathcal{C}$ in a descending order, by a *gain score* $gain(c)$, that combines: (1) the *attribute score c.S* (Sect. 5.5.1); and (2) the *attribute memory cost.* The latter formulates the memory overhead, when c is included in a tree. Since the memory cost of a tree depends on the number of distinct values of its attributes, we consider the domain size $|dom(c)|$ to quantify each attribute's memory cost.

We define the *gain score* of an attribute c as: $gain(c) = \frac{c.S}{|dom(c)|}$.

Given a gain-ordered list L_g of attributes $\mathcal{C}$, the *candidate tree set* $\mathcal{H}$, is defined by $|\mathcal{C}|$ trees, where each tree $h_i \in \mathcal{H}$ contains the first $(i+1)^{th}$ attributes of L_g. Therefore, the *candidate tree set* is $\mathcal{H} = \{h_0, ..h_{|\mathcal{C}|-1}\}$, with $h_i.\mathcal{C} = \{L_g[0], ...L_g[i]\}$. The *computational cost* for generating the candidate tree set, employing a linearithmic sorting algorithm (e.g., mergesort) is $O(|\mathcal{C}|\, log|\mathcal{C}|)$.

The candidate tree set can be characterized as a *small number of trees*, where each of them has *different memory cost* (i.e., number of attributes), while containing *as many "promising" attributes as possible.* The proposed algorithms consider only the candidate trees in the assignment selection process. This way, we reduce the $2^{|\mathcal{C}|}$ possible trees we have to examine to $|\mathcal{C}|$, significantly pruning the search space of the SIN problem.

Example 12. [***Candidate Trees***] From Example 9 we have the *attribute scores:* $A_P.S = 0.8$, $A_B.S = 0.8$, and $A_N.S = 0.1$. Also, we have the the following *domain sizes*: $|dom(P)| = 2$, $|dom(B)| = 4$, and $|dom(N)| = 3$. Hence, the *attributes gain scores* are $gain(P) = 0.8/2$, $gain(B) = 0.8/4$, and $gain(N) = 0.1/3$. Based on the gain scores, the sorted list of attributes is $L_g = \{P, B, N\}$. Thus, the *candidate trees* are $\mathcal{H} = \{h_P, h_{P,B}, h_{P,B,N}\}$, where $h_P.\mathcal{C} = \{P\}$, $h_{P,B}.\mathcal{C} = \{P, B\}$, and $h_{P,B,N}.\mathcal{C} = \{P, B, N\}$.

Tile-Tree Assignment Cost Estimation. The tile-tree assignment cost denotes the memory allocated by the assignment's tree. Recall from Section 5.2 that, a tree is populated during the initial file parsing, with the distinct values that appear in the categorical attributes of the data objects it contains. Therefore, the actual tree size is not known a priori, and should be estimated during index initialization.

As estimation we consider the worst case (i.e., the maximum memory a tree can require), that is defined by the maximum number of nodes the tree can have (see *Tree space complexity analysis* in Sect. 5.2.3). Let $nodes_{max}(\nu, \mathcal{C})$ denote the maximum number of nodes of a tree that contains $\mathcal{C}$ attributes and ν objects.

Assuming a uniform distribution of objects over the tiles, the estimated number of nodes per tile is $\nu_t = |\mathcal{O}_{DS}| \cdot \frac{areaSize(t)}{areaSize(DS)}$, where $areaSize(DS)$ and $areaSize(t)$ are the sizes of the 2D areas defined by the dataset objects (i.e., grid area size $|dom(A_x)| \cdot |dom(A_y)|$), and a tile t, respectively; and $|\mathcal{O}_{DS}|$ is the number of objects in the dataset. So, the *maximum cost estimation* for an assignment π_t^h is $\pi_t^h.\Phi = nodes_{max}(\nu_t, \mathcal{C}) \cdot n_{cost}$, where n_{cost} is the memory allocated by a single node.[3]

Eviction Mechanism. During both the assignment selection process and subsequent user exploration, the memory allocated for constructing and storing trees, tiles, and statistics may exceed the initial estimates or the available memory. To handle these cases, we introduce an eviction mechanism.

In the event that the memory demand exceeds the allocated budget, some trees need to be removed from memory. Our eviction policy is centered around tile utility value ρ_t. Specifically, the tree corresponding to the tile with the lowest utility is selected for eviction. This eviction process involves erasing the tree's structure (i.e., nodes, edges, and metadata) and reassigning its object entries to a single root node attached to the respective tile.

This policy does not involve writing trees to the disk. However, when the memory consumed by object entries exceeds the memory budget, the eviction mechanism presented in Section 4.5 is employed. This mechanism writes a tile's object entries to the disk and fetches them back into memory when a future query overlaps with that tile.

[3] Recall that, the memory for each node is (almost) the same, with the exception of the leaf nodes where metadata is stored. For simplicity, we assume that all nodes have equal memory size.

Algorithm 5. GRD ($\mathbb{I}_{\mathcal{T}}, \mathcal{A}_{\mathcal{C}}, \mathcal{B}_{\Pi}$)

```
Input: $\mathbb{I}_{\mathcal{T}}$: initialized tiles;   $\mathcal{A}_{\mathcal{C}}$: categorical attributes;
       $\mathcal{B}_{\Pi}$: memory budget for trees
Output: $\mathbb{I}_{\Pi}$: selected tile-tree assignments list
Variables: $W_{\pi}$: assignments list max-heap;
           $Cost_{\Pi}$: selected assignments appr. cost

1  $\mathcal{H} \leftarrow$ generateCandTrees($\mathcal{A}_{\mathcal{C}}$)                //generate candidate trees
2  foreach $(t, h) \in \mathbb{I}_{\mathcal{T}} \times \mathcal{H}$ do           //generate assignments & compute utilities
3      compute $\pi_t^h.\omega$ and $\pi_t^h.\Phi$           //assignment utility (Eq.5.2) & appr. cost (Sect.5.5.1)
4      $\pi_t^h.score \leftarrow$ assgnScore($\pi_t^h.\omega$, $\pi_t^h.\Phi$)   //compute assignment score   w.r.t. assignment's utility $\pi_t^h.\omega$ and appr. cost $\pi_t^h.\Phi$
5      push $\pi_t^h$ to $W_{\pi}$                               //initialize assignments max-heap
6  $Cost_{\Pi} \leftarrow 0$;
7  while $Cost_{\Pi} < \mathcal{B}_{\Pi}$ and $W_{\pi} \neq \varnothing$ do                   //select assignments
8      $\pi_{t_\gamma}^{h_\gamma} \leftarrow pop(W_{\pi})$                 //select (and remove) the top assignment
9      insert $\pi_{t_\gamma}^{h_\gamma}$ into $\mathbb{I}_{\Pi}$     //the selected assignment is inserted into assignments list
10     $Cost_{\Pi} \leftarrow Cost_{\Pi} + \pi_{t_\gamma}^{h_\gamma}.\Phi$
11 return $\mathbb{I}_{\Pi}$
```

5.6.2 Greedy Tile-Tree Assignments Algorithm (GRD)

Here we present a greedy algorithm (GRD) that finds the tile-tree assignments. The basic idea is that we first compute a utility score for each candidate assignment between a tree and a tile. All assignments are sorted in descending order based on their score. The algorithm selects the top assignments and aggregates their cost up to the one for which the total estimated cost is lower than the budget.

Algorithm Description. Algorithm 5 presents the pseudocode of GRD. GRD first generates the candidate tree set $\mathcal{H}$, using the generateCandTrees function (*line* 1). For each tile $t \in \mathbb{I}_{\mathcal{T}}$ and candidate tree $h \in \mathcal{H}$ (*loop in line* 2), the algorithm defines the assignment π_t^h, computes the assignment's utility $\pi_t^h.\omega$, and the assignment's estimated cost $\pi_t^t.\Phi$ (*line* 3).

Using these metrics, the function assgnScore computes the assignment score $\pi_t^h.score$, which increases w.r.t. assignment utility and decreases w.r.t. assignment cost (*line* 4). Formally, let x_1 and x_2 assignments utilities, and y_1 and y_2 assignments costs, then: $\text{assgnScore}(x_1, y_1) \geq \text{assgnScore}(x_2, y_1) \Leftrightarrow x_1 \geq x_2$, and $\text{assgnScore}(x_1, y_1) \geq \text{assgnScore}(x_1, y_2) \Leftrightarrow y_1 \leq y_2$.

Next, the assignment is inserted (using the *push* operation) into a max-heap W_{π} that sorts the assignments in descending order based on $\pi_t^h.score$ (*line* 5).

Next, GRD selects assignments as far as the total estimated cost Π_{cost} for the selected assignments is lower than the memory budget $\mathcal{B}_{\Pi}$, and the heap is not empty (*loop in line* 7). The assignment $\pi_{t_\gamma}^{h_\gamma}$ which has the largest score is selected and removed from the heap via the *pop* operation (*line* 8). Next, $\pi_{t_\gamma}^{h_\gamma}$ is inserted into the selected assignments list $\mathbb{I}_{\Pi}$ (*line* 9) and the estimated cost is updated (*line* 10). Obviously, if an assignment for a tile t is selected, the rest of the assignments referring to t are not examined.

Example 13. [***GRD Algorithm***] In this example, we assume that the estimated cost $\pi_t^t.\Phi$ of an assignment is equal to the cost presented in Table 5.2. Also, the assignment score is equal to assignment utility presented in Table 5.2. Finally, as in Example 11, we assume a budget of 50.

Algorithm 6. BINN ($\mathbb{I}_\mathcal{T}, \mathcal{A}_\mathcal{C}, \mathcal{B}_\Pi$)

```
Input: $\mathbb{I}_\mathcal{T}$: initialized tiles;   $\mathcal{A}_\mathcal{C}$: categorical attributes;
       $\mathcal{B}_\Pi$: memory budget for trees
Parameters: BS: binning strategy;   AI: assignments initialization strategy;
            TS: tree selection strategy
Output: $\mathbb{I}_\Pi$: selected tile-tree assignments list
Variables: $L_\mathcal{I}$: list of bins' intervals;   $L_\mathcal{T}$: list of tile sets per bin;   $L_\mathcal{H}$: list of selected trees for the
           tiles of each bin;
           $\mathcal{H}$: candidate trees;   $Cost_\Pi$: selected assignments appr. cost

1  $L_\mathcal{I} \leftarrow$ BS.determineBinsIntervalsOverTilesProb($\mathbb{I}_\mathcal{T}$)   //intervals are defined over tiles' probabilities $\rho_t$; $L_\mathcal{I}[i]$ is the interval of $i^{th}$ bin; intervals $L_\mathcal{I}$ are in ascending order
2  $L_\mathcal{T} \leftarrow$ group tiles $\mathbb{I}_\mathcal{T}$ into bins based on intervals $L_\mathcal{I}$   //$L_\mathcal{T}[i]$ is the set of tiles contained in the bin $i$ that is defined by the interval $L_\mathcal{I}[i]$
3  $\mathcal{H} \leftarrow$ generateCandTrees($\mathcal{A}_\mathcal{C}$)   //generate candidate trees
4  $L_\mathcal{H}[i] \leftarrow \varnothing$   $0 \le i \le |L_\mathcal{I}| - 1$   //selected trees list; $\mathcal{L}_\mathcal{H}[i]$ contains the tree selected for the tiles $L_\mathcal{T}[i]$ of the bin $i$
5  $Cost_\Pi \leftarrow 0$   //selected assignments appr. cost
6  if AI is $defined$ then   //an assignments initialization strategy has been defined
7      for $i \leftarrow 0$ to $|L_\mathcal{I}| - 1$ do   //assignments initialization – assign initial trees to bins
8          $L_\mathcal{H}[i] \leftarrow$ AI.selectInitialTreeForBin($i, L_\mathcal{H}, L_\mathcal{T}, \mathcal{H}, \mathcal{B}_\Pi, Cost_\Pi$)
9          $Cost_\Pi \leftarrow Cost_\Pi$ + assignmentsCostInBin($L_\mathcal{T}[i], L_\mathcal{H}[i]$)
10         if $Cost_\Pi \ge \mathcal{B}_\Pi$ then break
11 for $i \leftarrow 0$ to $|L_\mathcal{I}| - 1$ do   //find trees for bins (and possibly update/replace the inital)
12     $L_\mathcal{H}[i] \leftarrow$ TS.selectTreeForBin($i, L_\mathcal{H}, L_\mathcal{T}, \mathcal{H}, \mathcal{B}_\Pi, Cost_\Pi$)
13     $Cost_\Pi \leftarrow Cost_\Pi$ + assignmentsCostInBin($L_\mathcal{T}[i], L_\mathcal{H}[i]$)
14     if $Cost_\Pi \ge \mathcal{B}_\Pi$ then break
15 for $i \leftarrow 0$ to $|L_\mathcal{I}| - 1$ do   //generate assignments
16     $\forall t \in L_\mathcal{T}[i]$ :   insert $\pi_t^{L_\mathcal{H}[i]}$ into $\mathbb{I}_\Pi$
17 return $\mathbb{I}_\Pi$
```

Initially, the algorithm computes the assignment scores for each tile (t_1 - t_6) and the candidate trees $\mathcal{H} = \{h_P, h_{P,B}, h_{P,B,N}\}$. Then, based on their score, the tile-tree assignments are sorted in descending order.

Then, the algorithm selects the assignment with the largest score, i.e., $\pi_{t_1}^{h_{P,B,N}}$. After this selection the assignments referring to tile t_1 are omitted. During the selection process, in each selection the algorithm ensures that the cost for the selected assignments does not exceed the available budget.

In the end, the algorithm selects the assignments $\mathbb{I}_\Pi = \{\pi_{t_1}^{h_{P,B,N}}, \pi_{t_2}^{h_{P,B}}, \pi_{t_3}^{h_P}, \pi_{t_4}^{h_P}, \pi_{t_5}^{h_P}, \pi_{t_6}^{h_P}\}$, in this order. The index utility $\Omega(\mathbb{I}_\Pi)$ for these assignments is 0.835 and the estimated construction cost 50.

Complexity Analysis.

The candidate trees require $O(|\mathcal{C}|\ log|\mathcal{C}|)$ (*line* 1). The first loop (*lines* 2-5) is executed $|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|$ times. The score (*lines* 4 & 5) is computed in constant time $O(1)$, and the *push* operation (*line* 5) is performed in $O(1)$, assuming that W_π is a Fibonacci max-heap. Thus, the loop cost is $O(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|)$. The second loop (*lines* 7-10), in the worst case is executed $|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|$ times. The insertion in a linked list is $O(1)$, and the amortized cost of each *pop* operation is $O(log(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|))$. Thus, the (amortized) complexity for the second loop is: $O(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|\ (log(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|) + 1)) = O(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|\ log(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|))$. Therefore, the overall (amortized) complexity for the GRD algorithm is: $O(|\mathcal{C}|\ log|\mathcal{C}| + |\mathbb{I}_\mathcal{T}|\,|\mathcal{C}| + |\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|\ log(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|)) = O(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|\ log(|\mathbb{I}_\mathcal{T}|\,|\mathcal{C}|))$.

5.6.3 Binning-Based Tile-Tree Assignment Algorithm (BINN)

In this section, we propose the *Binning-Based Tile-Tree Assignment* algorithm (BINN). The basic idea of BINN is that the tiles are organized into bins, and the same candidate tree is assigned to every tile belonging to the same bin.

BINN Basic Characteristics. (1) The *bin-based tile organization* phase, in which the tiles are grouped into bins. The tree assignments are defined at bin-level and, thus, are not "strictly" affected by tile-specific factors, which in many cases may not be accurately estimated, such as the tile probability and the tile-tree assignment cost.

(2) The *assignment initialization phase*, which defines "default" assignments for (some) tiles. These assignments may be updated/replaced during the assignment selection process. Hence, this phase enables the algorithm to "impose" assignments to a set of tiles and/or "influence" the assignment selections that follow. For example, BINN may assign a tree with one attribute to the tiles with probability larger than a threshold, or to the top-k tiles.

(3) The *assignment selection phase*, which traverses and assigns a tree to each bin, by considering the selected and the default assignments in the rest of the bins.

BINN vs. GRD. BINN tackles a shortcoming of the GRD algorithm. Particularly, GRD allocates most of the budget assigning trees with all categorical attributes included (Fig. 5.9). As a result, trees are assigned to a smaller number of tiles. On the other hand, the bin-based approach adopted by the BINN algorithm leads to a more "balanced" allocation of the budget, with more tiles being assigned with trees having fewer categorical attributes. As demonstrated (Sect.5.7), in many cases, the small number of trees assigned by GRD compared to BINN has great impact in algorithms performance. In general, BINN is more than 1.5× faster and perform the half I/Os compared to GRD. To also remark that, in several cases BINN is more than 100× faster (Fig. 5.12).

Algorithm Description. Algorithm 6 presents the pseudocode.

Using the binning strategy BS, the algorithm determines the bins as a list of probability intervals $L_{\mathcal{I}}$, which are defined based on the probabilities of the input tiles $\mathbb{I}_{\mathcal{T}}$ (*line* 1). Then, tiles are inserted into the list $L_{\mathcal{T}}$ (*line* 2) and the candidate trees $\mathcal{H}$ are generated (*line* 3).

In the next step, if an assignment initialization strategy AI has been defined (*line* 6), the algorithm performs the assignments initialization phase.

For each bin i (*loop in line* 7), function selectInitialTreeForBin determines the default tree $L_{\mathcal{H}}[i]$ of the i^{th} bin (*line* 8). Next, function assignmentsCostInBin computes the cost of this assignment, considering the assigned tree $L_{\mathcal{H}}[i]$ for the tiles $L_{\mathcal{T}}[i]$ of bin i (*line* 9).

In the assignment selection phase (*loop in line* 11) and based on the tree selection strategy TS, the selectTreeForBin assigns one of the candidate trees $\mathcal{H}$ to bin i (*line* 12). In cases where an initialization phase is performed, the default tree may be replaced by the selected ones. Finally, it generates the tree assignments $\pi_t^{L_{\mathcal{H}}[i]}$ (*loop in line* 15) based on the tree $L_{\mathcal{H}}[i]$ selected for each bin i.

Strategies Details. Without loss of generality, in our experiments, the binning strategy BS uses equal frequency binning to define the bin intervals. The selectInitialTreeForBin (*line* 8) and selectTreeForBin (*line* 12) functions may consider several factors such as:

the already assigned trees $L_{\mathcal{H}}$ (assigned either during initialization, or during selection), the currently available budget ($\mathcal{B}_{\Pi} - Cost_{\Pi}$), the distribution of tile probabilities, etc.

In our implementation, we define a simple selectTreeForBin function, which assigns to each bin the candidate tree $\mathcal{H}[k]$ with the larger number of attributes, such that the cost of already selected and initialized assignments is lower than the budget. So, the function selectTreeForBin for a bin i selects:

$$\mathcal{H}[k] \quad \text{s.t.} \quad \arg\max \mathcal{H}[k] \quad \text{and}$$
$$Cost_{\Pi} + \mathsf{assignmentsCostInBin}(L_{\mathcal{T}}[i], \mathcal{H}[k]) < \mathcal{B}_{\Pi}.$$

Example 14. [***BINN Algorithm***] As in the previous example, we assume that the estimated assignment cost and score are equal to the cost and the utility presented in Table 5.2, and the budget is equal to 50. We adopt an equal frequency binning strategy to define the bin intervals (e.g., we consider two bins), and as selectTreeForBin we use the method described above.

Based on the tile utilities shown (in parenthesis) in Table 5.2, the following bins of tiles are defined: $L_{\mathcal{T}} = \{\{t_1, t_2, t_3\}, \{t_4, t_5, t_6\}\}$.

First, we consider the case where no assignment initialization strategy AI is used. Then, the list of trees selected for the bins is: $L_{\mathcal{H}} = \{h_{P,B}, h_P\}$, i.e., $h_{P,B}$ selected for the first bin. Finally, the tile-tree assignment set selected by the algorithm is: $\mathbb{I}_{\Pi} = \{\pi_{t_1}^{h_{P,B}}, \pi_{t_2}^{h_{P,B}}, \pi_{t_3}^{h_{P,B}}, \pi_{t_4}^{h_P}, \pi_{t_5}^{h_P}, \pi_{t_6}^{h_P}\}$, which results in a total index utility $\Omega(\mathbb{I}_{\Pi})$ equal to 0.85 and total estimated cost 33. Considering an AI strategy which pre-assigns a default $h_{P,B}$ to every tile, the final assignments become $\mathbb{I}_{\Pi} = \{\pi_{t_1}^{h_{P,B}}, \pi_{t_2}^{h_{P,B}}, \pi_{t_3}^{h_{P,B}}, \pi_{t_4}^{h_{P,B}}, \pi_{t_5}^{h_{P,B}}, \pi_{t_6}^{h_P}\}$, which result in total index utility $\Omega(\mathbb{I}_{\Pi}) = 0.9$ and total estimated cost 47.

Complexity Analysis. To determine the intervals of the bins (*line* 1) adopting a simple binning method (e.g., equal width/ frequency binning) can be performed by sorting (e.g., mergesort) and traversing once the tiles list. That is performed in $O(|\mathbb{I}_{\mathcal{T}}|\, log|\mathbb{I}_{\mathcal{T}}| + |\mathbb{I}_{\mathcal{T}}|)$. Then, in the worst case, organizing the tiles into bins (*line* 2) are performed in $O(|\mathbb{I}_{\mathcal{T}}|)$. The candidate trees require $O(|\mathcal{C}|\, log\, |\mathcal{C}|)$ (*line* 3).

We can easily verify that a large number of "rational" selectInitialTreeForBin (*line* 8) and selectTreeForBin (*line* 12) functions, cost $O(|L_{\mathcal{T}}[i]||\mathcal{C}|)$ in order to select a tree for bin i. In each selection, these functions examine the candidate trees CT, and in the same time compute the different costs. Since, in such functions the cost is computed during the selection, the function assignmentsCostInBin is omitted. Note that, the same complexities also hold in the functions used in our implementation. Thus, in the worst case, the *loop in line 7* costs $O(|\mathbb{I}_{\mathcal{T}}||\mathcal{C}|)$; the same also holds for the *loop in line 11.*

In the last loop in the worst case, the insert operation (*line* 16) is executed $|\mathbb{I}_{\mathcal{T}}|$ times. Thus, the cost of the insertions in the linked list is $O(|\mathbb{I}_{\mathcal{T}}|)$.

Therefore, in the the worst case the complexity of BINN is the sum of the aforementioned steps: $O(|\mathbb{I}_{\mathcal{T}}|\, log|\mathbb{I}_{\mathcal{T}}| + |\mathbb{I}_{\mathcal{T}}| + |\mathbb{I}_{\mathcal{T}}| + |\mathcal{C}|\, log|\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}||\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}||\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}|) = O(|\mathbb{I}_{\mathcal{T}}|\, log|\mathbb{I}_{\mathcal{T}}| + |\mathcal{C}|\, log|\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}|\, |\mathcal{C}|)$.

Table 5.3: VETI Evaluation: Datasets

Name	#Object	#Attributes	#Cat. Attributes	Raw Data File Size (GB)
Real Datasets				
TAXI	165M	18	5	26
NET	40M	150	6	45
Synthetic Datasets				
SYNTH10	100M	10	6	11
SYNTH50	100M	50	15	51

5.7 Experimental Analysis

The objective of our evaluation is to assess the performance of our approach in terms of time and number of I/Os. We evaluate different VETI variations and several competitors over two real and two synthetic datasets. In what follows we outline the key findings of our experiments. Following that, we provide the characteristics of the experimental setup and the detailed results of our evaluation study.

5.7.1 Results Highlights

(1) *Performance Overview*: In most queries, VETI exhibits *response time less than* 0.04sec, over large raw files (e.g., 45GB). Regarding the best of the examined systems, in most queries VETI *is up to 100× faster* and performs up to *2 orders* of magnitude fewer I/O operations.
(2) *Data Characteristics*: All VETI variations report (sub-)linear performance w.r.t. the number of objects and categorical attributes, as well as the domain size.
(3) *VETI Variations*: Regarding the VETI variations, both VETI-BINN and VETI-GRD outperform the naive VETI-RND. VETI-BINN is more than 1.5× faster and requires about half the I/O operations compared to VETI-GRD. VETI-BINN performs even better when the user moves further away from the initial query and/or when the initialization budget is small.
(4) *Initialization Phase*: In the initialization phase, VETI-BINN is on average 8× faster than MySQL, 1.2× faster than PostgresRaw, and slightly slower than VALINOR.

5.7.2 Experimental Setup

5.7.2.1 Datasets & Queries

In our experimental evaluation of VETI, we have used two *real datasets*, the *NYC Yellow Taxi Trip Records* (TAXI) and *a telecommunication network quality dataset* (NET); and two synthetic ones (SYNTH10 / 50).[4] Table 5.3 presents the basic characteristics of the datasets used.

Queries Template. Each query contains: (1) a *Select* clause defined over the axis-attributes; (2) a *Group-by* clause on a categorical attribute; (3) a *Filter* clause that contains either 1 or 2 equality conditions, specified over randomly selected categorical attributes and values from their corresponding domains; and (4) an *Analysis*

[4] The data generator and the queries are available at: github.com/VisualFacts/RawVis

clause that computes 5 aggregate functions over a numeric attribute, i.e., min, max, std, variance, and mean.

TAXI Real Dataset. The TAXI dataset is a CSV file, containing information regarding taxi rides in NYC.[5] Each record corresponds to a trip, described by 18 *attributes*. From these attributes, 5 are categorical: Payment Type, Passenger Count, Rate Type, Provider Code and Store & Forward Flag. We selected a subset of this dataset for 2014 trips with 165M objects and 26 GB CSV file size.

The Longitude and Latitude of the pick-up location are the *axis attributes* of the exploration. The *Select* clause is defined over an area of 2km × 2km size, with the *first query* Q_0 posed in central Manhattan. The *Group-by* clause contains the Passenger Count attribute, and the *Analysis* clause the Tip Amount.

NET Real Dataset. The second *real dataset* (NET) is an anonymized proprietary *telecommunication quality network dataset* containing latency and signal strength measurements crowdsourced from mobile devices in the Greater Tokyo Area (40M objects, 45GB csv file). Each record is described by 150 *attributes*. We selected the *categorical attributes*: Network Type (e.g., 4G), Network Operator Name, Device Manufacturer, Location Provider, *OS version*, and Success Flag.

The Latitude and Longitude are the *axis attributes*, simulating a map-based exploration scenario, starting from central Tokyo. The *Select* clause is defined over a 4km × 4km area. The *Group-by* clause contains the Network Type, and Latency is used in the *Analysis* clause.

SYNTH10 / 50 Synthetic Datasets. Regarding the *synthetic datasets* (SYNTH10 / 50), we have generated two CSV files of 100M *data objects* (in the default setting), having 10 and 50 *attributes* (11 and 51 GB, respectively). The datasets contain *numeric attributes* in the range [0, 1000], as well as *categorical attributes*, where the values of the numeric and categorical attributes follow a uniform distribution. In our experiments, we vary the *number of objects* from 1M to 500M, objects with 100M being the default value, where the size of the dataset having 500M objects is 52GB. Regarding queries, as in 4.6, the *Select* clause is defined over two numeric attributes that specify a window size containing approximately 100K objects.

5.7.2.2 Exploration Scenario

In our evaluation, we consider a typical exploration scenario in which the user explores different areas, and also filters, and performs Group-by operations. We have generated sequences of 100 overlapping queries, with each window query shifted (i.e., pan operation) in relation to its previous one by 10% towards a random direction.

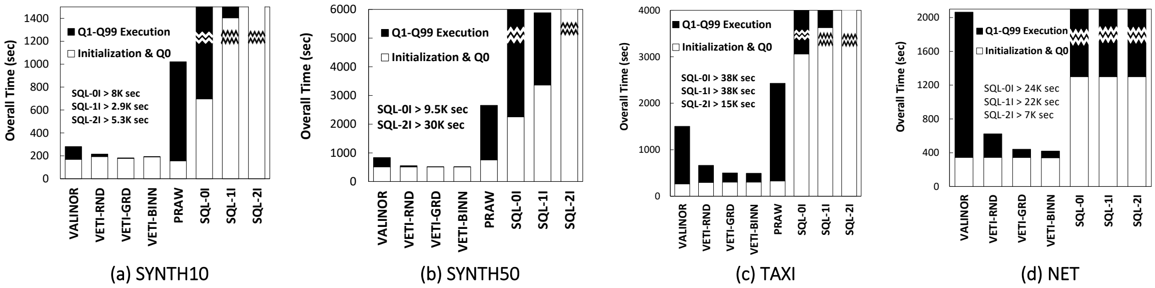
This scenario attempts to formulate a common user behavior in 2D visual exploration, where the user explores nearby regions using pan and zoom operations [102, 54, 89, 12, 96, 26], such as the "region-of-interest" or "following-a-path" scenarios which are commonly used in map-based visual exploration.

Additionally, to formulate the "repetitive calculation of statistics" that commonly appears in exploration scenarios (Sect. 5.5.1) [94], we included the attributes of the initial query in the generated sequence of queries four times more frequently than the other dataset attributes.

[5] www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

Table 5.4: VETI Evaluation: Basic Parameters

Description	Values
Synthetic Datasets	
Number of Objects (Millions)	5, 10, 50, **100M**, 200, 500
Number of Categorical Attributes	3, 4, **6**, 10, 15
Categorical Attribute Domain Size	5 **10**, 20, 50
Synthetic & Real Datasets	
Budget Size (GB)	0.5, 1, **2**, 3, 5
Number of Bins	50, **100**, 500, 1000



(a) SYNTH10 (b) SYNTH50 (c) TAXI (d) NET

Figure 5.5: Overall Time (Broken down to Initialization & Q_1~Q_{99} Evaluation Time)

5.7.2.3 VETI Parameters

Regarding VETI's tile structure, we adopt the setting used in 4.6, where the tile structure is initialized with 100 × 100 equal-width tiles, while an extra 20% of the number $|\mathcal{T}_0|$ of initial tiles was also distributed around the first query Q_0 using the Query-driven initialization method (Sec. 4.4.1). Also, the numeric threshold for the adaptation of VETI was set to 200 objects.

The index initialization budget, is varied from 0.5 to 5GB, with 2GB being the default value. Recall that this memory budget includes only the memory allocated by the tile and tree structures, and does not include the memory required to store the object entries.

5.7.2.4 VETI Variations

We evaluate two versions of VETI, named VETI-GRD and VETI-BINN, based on the GRD and BINN algorithms (Sect.5.6). Moreover, we consider a naive assignment approach, titled VETI-RND, which follows a random tile-tree assignment strategy. It first sorts the tiles based on the tile probability ρ_t, then assigns a randomly selected tree from the entire powerset $HP_{\mathcal{C}}$ to each tile, until the budget constraint $\mathcal{B}$ is satisfied.

5.7.2.5 Competitors

We compare our method with: (1) VALINOR which contains only the tile-based indexing structure without the CET index; (2) A traditional DBMS (MySQL 8.0.22), where data is loaded and indexed in advance; three indexing settings are considered: (*a*) no indexing (SQL-0I); (*b*) one composite B-tree on the two axis attributes (SQL-1I); and (*c*) two single B-trees, one for each of the two axis attributes (SQL-2I).

MySQL also supports SQL querying over external files (see CSV Storage Engine in Sec. 2.2); however, due to low performance [7], we do not consider it as a competitor. (3) PostgresRaw (PostgresRaw)[6], built on top of Postgres 9.0.0 [7], which is a generic platform for in-situ querying over raw data (Sect. 2.2). Note that, due to parsing/processing problems on the NET dataset with the PostgresRaw, we did not manage to load and report experiments on this combination.

5.7.2.6 Metrics

In our experiments, we measure the: (1) *Evaluation Time* of a query; (2) *Initialization Time*, which corresponds to the time required to initialize the index and return the results of the first query Q_0, i.e., from-raw data-to-1st result time. Regarding the initialization phase of the examined systems we have: (a) before evaluating Q_0, MySQL needs to parse the raw file, load, and index (except SQL-0I) the data; (b) during evaluating Q_0, PostgresRaw needs to parse the raw file and construct the positional map; (c) during evaluating Q_0, VALINOR parses the raw file, generates the tile index structure, and populates it with the object entries; and (d) during evaluating Q_0, beyond the actions performed by VALINOR, VETI also parses the categorical attributes and constructs the tree indexes over the tiles. (3) *Overall Execution Time* of an exploration scenario, that includes: initialization time and query evaluation time for all the queries included in the exploration scenario, i.e., workload; (4) *I/O Operations* performed during query evaluation (for I/O definition see Sect. 5.3.2); and (5) *Index Utility.* Table 5.4 summarizes the parameters that we vary in the experiments.

5.7.2.7 Implementation

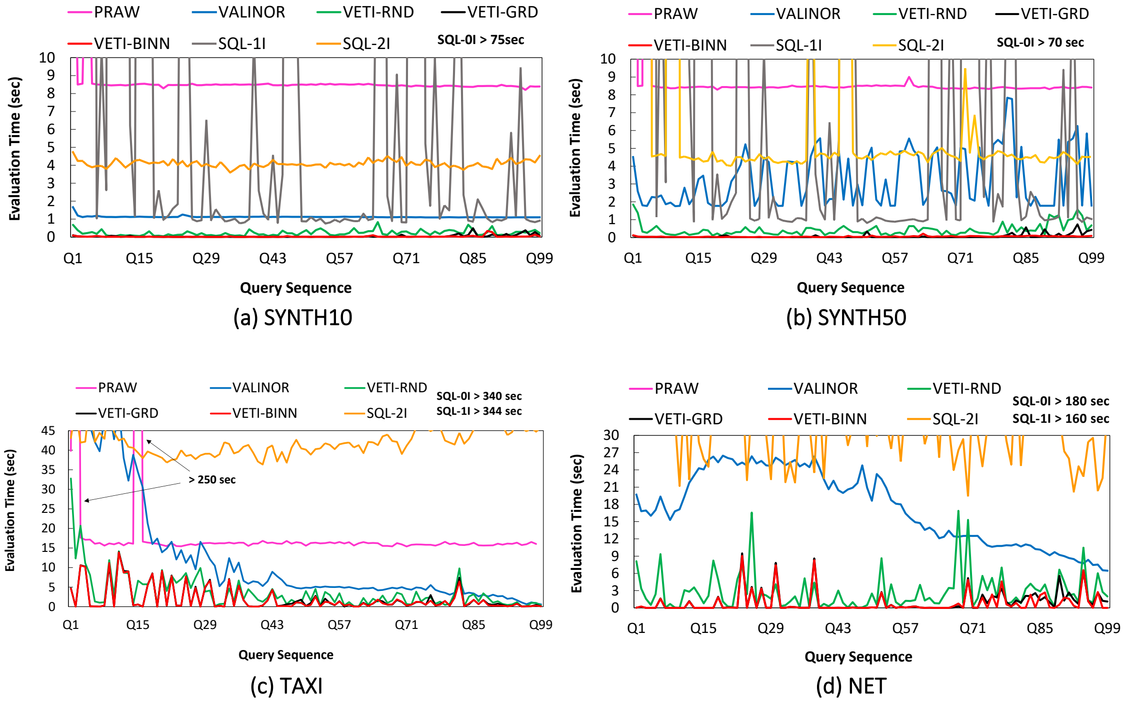
VETI is implemented on JVM 1.8 and the experiments were conducted on an 3.60GHz Intel Core i7-3820 with 64GB of RAM. We applied memory constraints (32GB max Java heap size) in order to measure the performance of our approach and our competitors. However, PostgresRaw required more than 32GB of memory for the synthetic datasets and more than 50GB for the TAXI dataset.

5.7.3 Performance

5.7.3.1 Initialization Phase: From-Raw Data-to-1st Result Time

Figure 5.5 presents the overall execution time which is split between the initialization time and the time for evaluating all the queries Q_1~Q_{99}. Recall that, the initialization time includes the time for parsing, loading the data (in the case of MySQL), constructing the index and answering the first query Q_0. In Figure 5.5 we can observe that the MySQL settings we examined exhibit the worst performance for evaluating Q_0, since MySQL needs to parse all attributes of the raw file and load the data in the disk. Also, for the SQL-1I and SQL-2I cases, the corresponding indexes must be built, which explains the increased initialization time in relation to SQL-0I where no index is generated. Both VALINOR and the VETI variations exhibit better initialization performance compared to PostgresRaw for the SYNTH50 and TAXI datasets, while for the SYNTH10 dataset, VETI requires

[6] https://github.com/HBPMedical/PostgresRAW



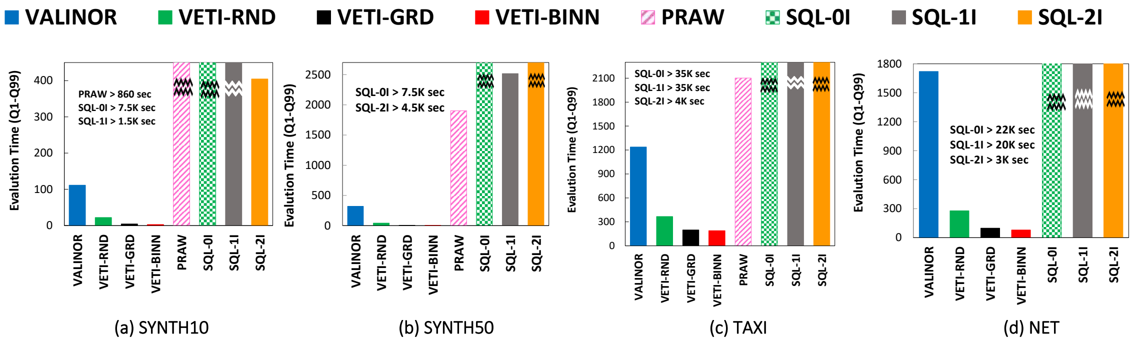
(a) SYNTH10

(b) SYNTH50

(c) TAXI

(d) NET

Figure 5.6: Evaluation Time per Query (sec)



(a) SYNTH10

(b) SYNTH50

(c) TAXI

(d) NET

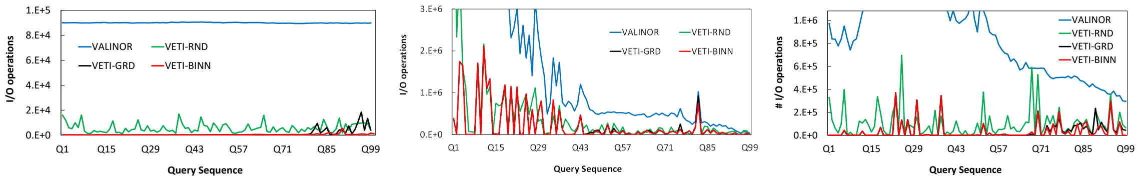
Figure 5.7: VETI Evaluation Time for Q_1~Q_{99} (sec)

a slightly higher initialization time. As it is expected, VETI variations are slightly slower during the initialization compared to VALINOR, since VETI needs to determine the tile-tree assignments, parse the categorical attributes, and create the tree structures. All VETI variations, however, exhibit similar initialization time, since the tile-tree assignment time is negligible compared to the time for parsing the file.

5.7.3.2 Evaluation Time per Query

Figure 5.6 presents the evaluation time for each individual query. Compared to the other methods, all VETI variations exhibit significantly lower evaluation time in almost all queries and datasets. In most queries, VETI reports *evaluation time less than* 0.04sec. On the other hand, the best competitors, PostgresRaw and VALINOR require for most queries more than 8 and 4sec, respectively. Overall, VETI *is more than 200× and 100× faster compared to* PostgresRawand VALINOR, respectively.

Regarding SQL, SQL-0I performs worse than the 3 SQL settings we examined, and requires approximately the same time for each query. This is expected as



(a) SYNTH10 (b) TAXI (c) NET

Figure 5.8: VETI: Number of I/Os per Query

SQL-0I has no index. From the other 2 settings, SQL-1I is for most queries faster than SQL-2I for the two synthetic datasets, and slower for TAXI and NET.

Regarding PostgresRaw, we observe that it exhibits a stable performance (after the first queries), which is however worse than both VALINOR and VETI in all datasets. The positional map used in PostgresRaw, attempts to reduce the parsing and tokenizing costs of future queries, by maintaining the position of specific attributes for every object in the raw file. However, PostgresRaw still needs to examine all objects in the dataset in order to select the ones contained in a 2D window query. Also, in contrast to VETI, PostgresRaw does not keep any metadata in order to efficiently compute the aggregate queries. Some of the early queries (approximately until Q_{15}) PostgresRaw exhibits noticeably higher time than the rest, and comparable to the time required to answer Q_0. This is due to the filter conditions of the queries. When a query refers to an attribute that was not included in Q_0, PostgresRaw needs to populate the positional map with it. In subsequent queries, which refer to indexed attributes, PostgresRaw exhibits a relatively constant evaluation time.

Regarding VALINOR, all variations of VETI report smaller evaluation time. Even though both VALINOR and VETI attempt to adapt to the workload and maintain metadata to speed up query evaluation time by reducing I/Os, VALINOR does not include any indexing capabilities for categorical attributes and thus it needs to access the file in order to evaluate queries with conditions to such attributes. In contrast, VETI variations exploit the tree organization for evaluating filter conditions on categorical attributes and the metadata stored in the leaves for evaluating the analysis and grouping operations of queries overlapping with fully contained tiles.

5.7.3.3 Evaluation Time for all Q_1~Q_{99} Queries

Figure 5.7 presents the evaluation time for the Q_1~Q_{99} queries. The behavior of the methods is similarly between the datasets, the variations of VETI significantly outperform the competitors. The best competitor, VALINOR needs about 30, 60, 7 and 20× more time for SYNTH10, SYNTH50, TAXI and NET, respectively, to evaluate all queries. Also, PostgresRaw is 270, 380 and 11× slower for SYNTH10/50 and TAXI, respectively.

5.7.3.4 I/O Operations

The evaluation time for VETI and VALINOR is mainly determined by the number of I/O operations. This can be observed in Figure 5.8, where the number of I/O operations per query exhibits approximately the same behavior with that of the evaluation time (Fig. 5.6). Note that we do not present the I/Os for PostgresRaw and

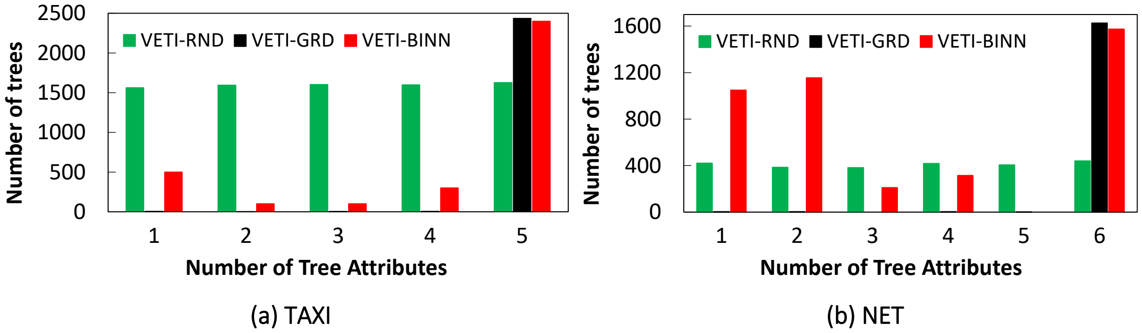


Figure 5.9: Number of Generated Trees vs. Number of Tree Attributes

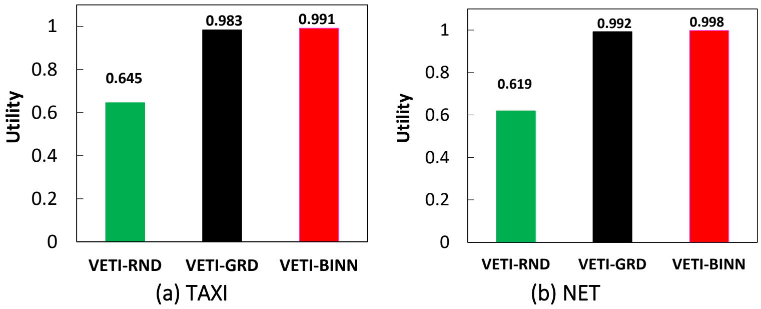


Figure 5.10: VETI Utility Score

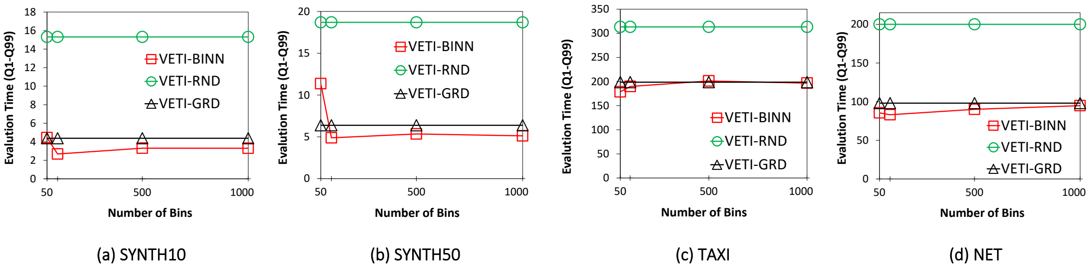
SQL, since they follow different workflows/methods for accessing the file, compared to our work. Also, the plot for SYNTH50 is omitted since it closely matches that for SYNTH10. Compared to VALINOR, the VETI variations perform up to 2 orders of magnitude less I/Os. This occurs since VALINOR has to access the raw file for every object contained in the 2D window query in order to retrieve the categorical attribute values required by the query.

5.7.4 VETI Variations

5.7.4.1 Performance & Assignments

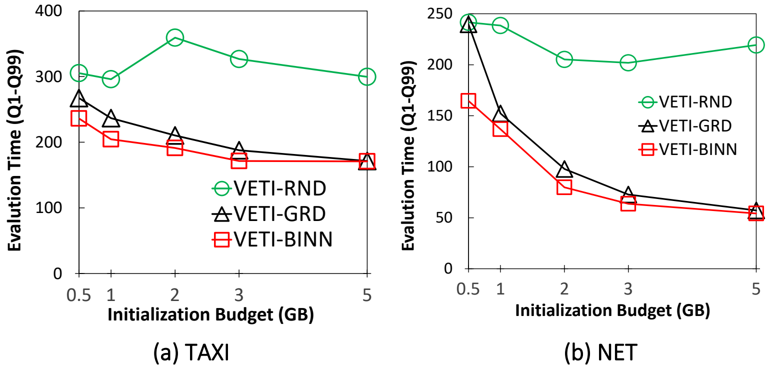
Here, we compare the performance of the three VETI initialization variations. Overall, considering the performance of VETI variations (Fig. 5.6), both VETI-GRD and VETI-BINN lead to faster query responses than the naive VETI-RND, in almost all cases. Also, VETI-BINN significantly outperforms the other two in the number of I/Os (Fig. 5.8). Considering the time required for all queries (Fig. 5.7), VETI-RND is on average 3× slower than VETI-GRD and VETI-BINN. Comparing VETI-GRD and VETI-BINN, VETI-BINN is more than 1.5× faster than VETI-GRD (in some cases more than 100× faster) (Fig. 5.7), and performs more than 3× less I/Os.

The difference in performance is the result of the different assignment policies (see Sect.5.6.3 for the policies used in VETI-BINN). Figure 5.9 depicts the number of trees generated during the initialization w.r.t. the number of attributes they have. For brevity, we omit the results for the synthetic datasets, as they exhibit similar behavior. We can observe, that VETI-RND follows a uniform distribution w.r.t. the number of tree attributes. In VETI-GRD the budget is mostly allocated at constructing trees that contain all of the categorical attributes. In contrast, VETI-BINN creates a more balanced distribution of the trees' number of attributes. As



(a) SYNTH10 (b) SYNTH50 (c) TAXI (d) NET

Figure 5.11: VETI-BINN: Evaluation Time for Q_1~Q_{99} (sec) vs. Number of Bins



(a) TAXI (b) NET

Figure 5.12: VETI Evaluation Time for Q_1~Q_{99} (sec) vs. Initialization Memory Budget

a result, VETI-GRD assigns "taller" trees to a smaller number of tiles. So, due to location-based assignments process we follow, the tiles located farther away from Q_0 tend to not contains trees. This is why, compared to VETI-GRD, VETI-BINN tends to perform even better when the user moves away from the initial starting point. This is demonstrated, in the query performance where, in most cases, after query Q_{75}, VETI-BINN is 10× faster than VETI-GRD in (Fig. 5.6); also, in some queries is up to 400× faster (Fig. 5.6d).

The impact of the different assignment strategies is also shown in the utility score (Fig. 5.10). Due to randomized tree assignments, VETI-RND results in a lower utility score, whereas VETI-BINN exhibits larger utility compared to VETI-GRD.

5.7.4.2 VETI-BINN: Varying the Number of Bins

In this experiment we study the performance of VETI-BINN w.r.t. the number of bins. Figure 5.11 presents the evaluation time for Q_1~Q_{99}, varying the number of bins from 50 to 1000. Note that, in the plots we include VETI-RND and VETI-GRD for the sake of comparison, even though they do not depend on the number of bins.

As we can observe, the performance of VETI-BINN is not highly affected by the number of bins, except for small number of bins, i.e., between 50 and 100. Based on our adopted assignment policies for VETI-BINN (Sect. 5.6.3), the following holds. For small numbers of bins, the assignment is more coarse-grained, i.e., shorter trees are assigned to the majority of the tiles. Increasing the number of bins results in more fine-grained assignments of trees to (bins of) tiles. However, trees that are assigned to bins near Q_0 will be taller, whereas the trees assigned to the remaining bins will be short. This explains why, in general, as the number of bins increase, the

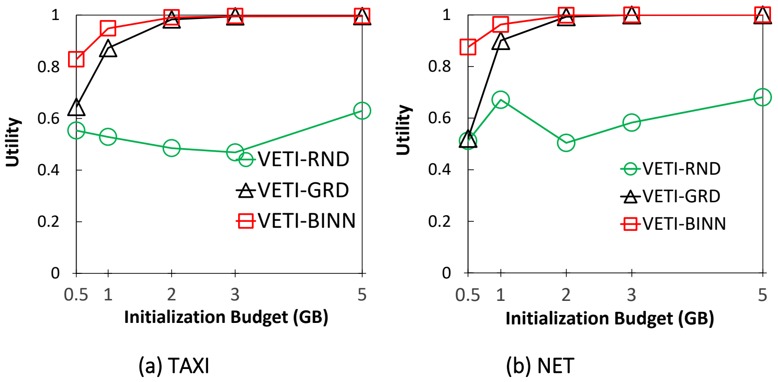


Figure 5.13: VETI Utility vs. Initialization Memory Budget

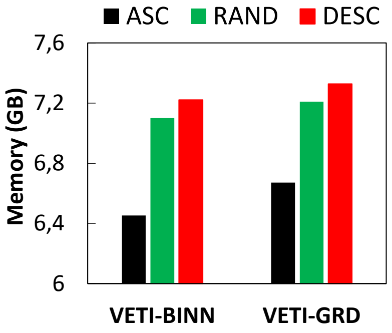


Figure 5.14: VETI Memory Size vs. Sorting Attributes based on Domain Size

performance of VETI-BINN approaches that of VETI-GRD. Recall that VETI-GRD assigns mostly tall trees (Fig. 5.9).

As previously mentioned, we should note that, the definition of bins depends on the dataset characteristics and the exploration scenario. As a general observation, in exploration scenarios with queries affecting areas away from the initial starting point, the number of bins should be kept relatively small in order to create trees (even short ones) to the majority of the tiles, whereas in scenarios focused on a specific area, increasing the number of bins performs better.

5.7.4.3 Varying the Initialization Memory Budget

In the first experiment, we evaluate the performance of VETI while varying the initialization budget from 0.5 to 5GB. Recall that this memory budget includes only the memory allocated by the tile and tree structures, and does not include the memory required to store the object entries. Note that, the plots for the SYNTH10/50 datasets are omitted since they are similar to the ones presented.

The evaluation time needed to evaluate all the queries is shown in Figure 5.12. The evaluation time decreases as the available memory budget increases. This is the result of the larger number and more detailed tree structures that are constructed with more budget, which leads to faster query evaluation. This is observed in both VETI-GRD and VETI-BINN. Regarding VETI-RND, its performance does not always improve when increasing the budget, as it allocates the budget in random tile-tree assignments.

Compared to VETI-GRD, VETI-BINN's performance is less dependent on the available budget. VETI-GRD performs much worse for low values of memory bud-

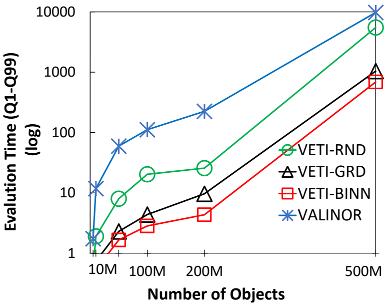
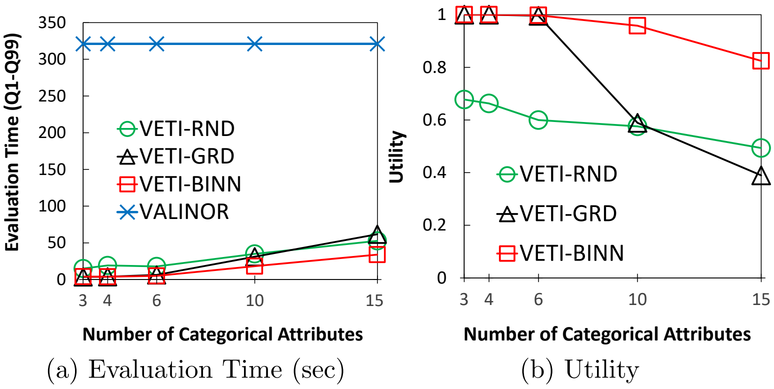


Figure 5.15: VETI Evaluation Time (log) vs. Number of Objects [SYNTH10]



(a) Evaluation Time (sec) (b) Utility

Figure 5.16: Varying the Number of Cat. Attributes [SYNTH50]

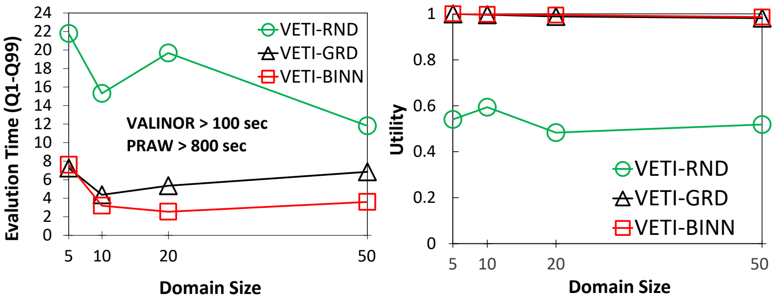
get, as it mostly assigns trees with all the categorical attributes which quickly depletes the budget on very few tiles. On the other hand, as the budget increases, the performance of VETI-GRD is comparable with that of VETI-BINN.

In more details, on small amounts of budget, VETI-BINN evaluates all the queries of the exploration scenario in half the time compared to VETI-GRD (Fig. 5.12). Moreover, at query level, in several queries, compared to VETI-GRD, VETI-BINN is from 100 to 3000× faster for the NET dataset; and more than 50× for SYNTH10/50.

In this experiment, we compute the utility score w.r.t. memory budget. (Fig. 5.13). The results closely match the evaluation time presented above. Specifically, the total index utility increases with higher budget for both VETI-GRD and VETI-BINN. Also, the utility of VETI-GRD is much lower than that of VETI-BINN for smaller amounts of budget, but their values converge as the budget increases.

5.7.4.4 VETI Memory Size vs. Sorting Tree Attributes

In this experiment, we examine how the sorting of tree attributes w.r.t. their domain size affects the allocated memory (more details in Sect. 5.2). In order to assess the effect of domain size, we create a version of the SYNTH10 where its categorical attributes had a different domain size, varying from 2 to 100. We measured the VETI memory size after initialization while sorting the attributes based on their domain sizes in ascending (ASC), descending (DESC), and random (RAND) order. As it can be seen in Figure 5.14, ASC ordering corresponds to the best case, while DESC to the worst. Specifically, for VETI-BINN the ASC ordering results in a decrease in memory size of around 10% and 8% in relation to DESC and RAND, respectively. Similar results are reported for VETI-GRD.



(a) Evaluation Time (sec) (b) Utility

Figure 5.17: Varying the Domain Size of Cat. Attributes [SYNTH10]

5.7.5 Effect of the Data Characteristics

5.7.5.1 Varying the Number of Objects

In this experiment, we evaluate the impact of the number of objects on the performance of VETI. For this, we vary the number of objects of SYNTH10 from 5 to 500M, and the evaluation time for Q_1~Q_{99} is presented in Figure 5.15. As the total number of objects in the file increase, the evaluation time increases (sub-)linearly for all variations of VETI as well as for VALINOR. This is reasonable considering that the index becomes more dense, the queries relatively select more objects and the number of required I/O operations increase; also, the cost of an I/O operation becomes "relatively" larger when the file size increase. Regarding VETI-RND, its performance is affected to a much greater extent compared to VETI-GRD and VETI-BINN, as its randomized tree assignment lead to a much higher I/O cost.

5.7.5.2 Varying the Number of Categorical Attributes

In this experiment, we vary the number of categorical attributes (Fig. 5.16). Here, we used the SYNTH50 dataset in order to be able to select up to 15 categorical attributes. For brevity the SQL and PostgresRaw methods are omitted, as they exhibit much higher evaluation time. Also, we could not evaluate PostgresRaw for 15 categorical attributes, due to increased memory requirements. Note that VALINOR's performance is not affected by the number of categorical attributes, since it does not consider them in its index structure.

As we can observe, query evaluation time increases for all methods, along with the number of attributes (Fig. 5.16(a)). VETI-BINN outperforms both VETI-RND and VETI-GRD. Regarding VETI-GRD, we can observe that it outperforms VETI-RND in every case except for the case of 15 categorical attributes. This is due to the fact that VETI-GRD allocates most of the budget for creating trees with all the categorical attributes. As a result, with a higher number of categorical attributes, VETI-GRD assigns trees to very few tiles, which explains its performance deterioration for 15 attributes. This is also depicted in Figure 5.16(b) which presents the utility score. As we can observe, in all VETI variations the utility score decreases as the number of categorical attributes indexed increase. This decrease is even more notable in the case of VETI-GRD, which after 10 attributes gets worse than both VETI-BINN and VETI-RND.

5.7.5.3 Varying the Domain Size of Categorical Attributes

In this experiment, we study the effect of the domain size. We generate 4 different versions of the SYNTH10 dataset, where the domain size of each categorical attribute for each one was set to 5, 10, 20 and 50. Note that, the results for the SYNTH50 are not presented since they are similar.

The evaluation time needed to execute the Q_1~Q_{99} is shown in Figure 5.17(a). Note that the plot shows only the VETI variations, since VALINOR does not depend on the domain, and others exhibit much higher evaluation time. We can observe that the evaluation time of VETI-GRD (resp. VETI-BINN) decreases from domain size 5 to 10 (resp. 20), and then increases.

This behavior is explained as follows. The attributes in the synthetic dataset have values, which are uniformly distributed over the objects. As the domain size of an attribute increases, the number of objects, which evaluate to the filter condition on this attribute, decreases, and so does the number of I/O operations. This explains the initial drop in query evaluation time for both VETI-GRD and VETI-BINN.

On the other hand, given the same number of attributes, a larger domain size results in trees with larger size in memory (Sect. 5.2.3). This explains the increase in evaluation time after domain size 10 for VETI-GRD and 20 for VETI-BINN, as the larger tree sizes result in fewer tiles getting assigned with trees.

5.8 Summary

In this chapter, we have explored the VETI index, our proposed solution for enabling efficient visual exploration and analysis of data existing in raw data fiels. The VETI index, as we have detailed, offers a significant expansion on the capabilities of the VALINOR index, focusing on the effective handling of categorical attributes often found in various visual techniques such as bar charts and heat maps.

We designed VETI as a hybrid main-memory indexing scheme that organizes data based on both numeric (or spatial) and categorical attributes. The index, which is built on-the-fly from the first user query, is designed to adapt progressively with user interactions and the specific types of analysis being conducted.

Recognizing the potential for high memory requirements, we proposed a resource-aware index initialization approach. This we formulated as an optimization problem and provided two approximate algorithms for its efficient solution.

To further optimize performance, we designed efficient query evaluation methods. By making effective use of the metadata stored in the index, these methods significantly reduce I/O operations, contributing to faster user response times.

In extensive experimental evaluations involving both real-world and synthetic datasets, VETI has shown significant superiority over existing solutions in terms of query response time and minimization of I/O operations.

In conclusion, the VETI index serves as a demonstration of how sophisticated indexing and adaptive query evaluation techniques can successfully address the challenges posed by large raw data files in visual explorations, particularly when dealing with categorical attributes.

Chapter 6

The RawVis Framework

6.1 Introduction

This chapter presents RawVis, an innovative tool that enables real-time visual exploration of raw data files. RawVis represents the culmination of concepts, techniques, and indexing schemes presented in previous chapters. It exemplifies how these principles can be combined into a coherent system to facilitate data exploration in an intuitive, interactive manner.

In particular, RawVis employs the exploration model and indexing schemes discussed in the previous chapters to enable efficient and interactive visual exploration and analysis of raw data files. This approach bypasses the need for data preprocessing, and instead, the system adjusts dynamically to the user's exploration patterns and the type of analysis performed. The result is a tool that offers a user-centric, responsive environment for data exploration.

The chapter is organized as follows:

- In Section 6.2, we discuss the architectural design of the RawVis system, providing an overview of its various components.
- Section 6.3 presents a detailed examination of the user interface components.
- Section 6.4 details the user study we conducted to evaluate the usability and performance of our tool.

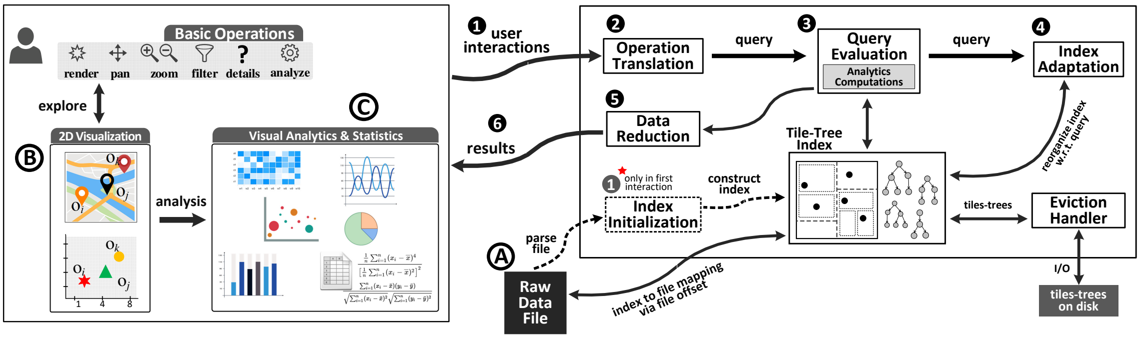


Figure 6.1: RawVis Architecture

6.2 System Overview

Figure 6.1 presents the architecture of the RawVis system; the frontend (web-based) is presented on the left side of the figure, and the backend on the right. In our working scenario, we consider that a *user visually explores* the data stored in a single *csv data file* Ⓐ in disk using a *2D visualization technique* (e.g., map, scatter plot) Ⓑ, and *analyzes it using visual* (e.g., bar and line charts, heatmaps, parallel coordinates), and *statistical* methods Ⓒ (e.g., Pearson correlation, covariance). Data attributes may be numeric, spatiotemporal, categorical, or textual; at least two of them must be numeric (e.g., longitude, latitude) and can be mapped to the X and Y axis of the 2D visualization.

In the backend, ❶ the *first time* the user requests to visualize or analyze a new dataset, the *file is parsed and indexed on-the-fly*, generating a "*crude*" initial version of the index (*Index Initialization* component). RawVis is built on top of the VETI index, combining the *tile-based multilevel structure* of VALINOR for organizing data into tiles for efficient exploration in the 2D plane; with the CET structure for further organizing a tile's objects based on its categorical values to offer efficient categorical-based group-by and filter operations.

In parallel with the index construction, the results corresponding to the first user request are evaluated. ❷ Based on the *exploration model* the user's visual and analytic operations (i.e., interactions) are *translated to data-access operations* (*Operation Translation* component), which are then ❸ evaluated over the *VETI Tile-Tree index* structure (*Query Evaluation* component) to compute and fetch the results. ❹ Based on the last user request, the index is *adapted progressively*, reorganizing its contents, and updating computed statistics (*Index Adaptation* component). ❺ The query results are further processed and reduced (e.g., via clustering, sampling, aggregation) before they are rendered in the visualization component, such that over-plotting issues are properly addressed (*Data Reduction* component). ❻ Finally, the results are returned and visualized to the user. Note that, during the index construction or the query evaluation, the index structure may not fit in main memory; in such cases, the *Eviction Handler* component stores parts of the index structure in the disk.

In the frontend, a 2D visual data representation is presented, as well as exploration operations, visual analytics and statistics (Sect. 6.3).

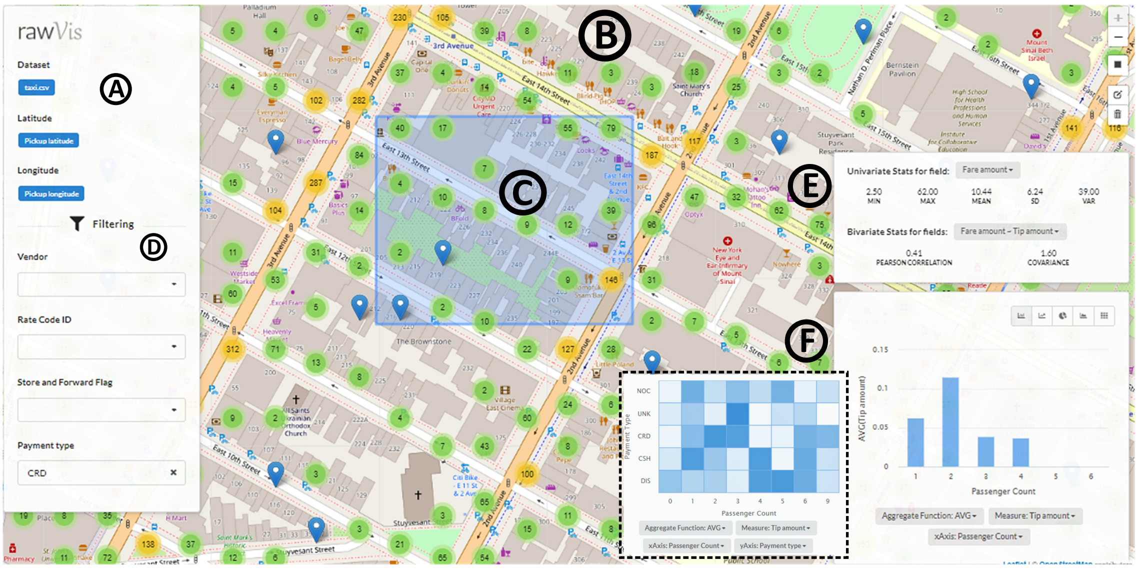


Figure 6.2: RawVis User Interface

6.2.1 Implementation Details.

RawVis is implemented on top of several open-source tools and libraries and is available under GNU/GPL.[1] The frontend was developed in TypeScript as a single-page application using the React library, while the backend was developed in Java 1.8. The frontend client app interacts with the backend with a REST API. For the visualization of the results the Leaflet and Highcharts libraries were used.

6.3 RawVis User Interface

This section briefly introduces the RawVis visual interface (Fig. 6.2). The tool is available at: *http://rawviz.imsi.athenarc.gr*. Also a video presenting the basic functionality of our prototype is available at: *https://vimeo.com/500596816*.

Figure 6.2 depicts pick-up points from the NYC Yellow Taxi Trip dataset[1], which is CSV files, containing information regarding yellow taxi rides in NYC. Each object refers to a specific taxi ride described by several attributes, such as pick-up location, trip distance, payment type, passenger count, tip amount.

6.3.1 UI Panels

The UI consists of the following panels:

- **Dataset Information Panel** Ⓐ: This panel allows the user to select the CSV file to explore. It presents details about the dataset, such as the axis attributes (latitude and longitude) that will be used for map-based exploration. Selecting a different CSV file loads its specific data attributes and updates the rest of the UI components.

[1] https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

- **Map Visualization Panel** Ⓑ: This panel offers map-based exploration, allowing the user to interact with the map through actions like zooming and panning.

- **Area Selection Panel** Ⓒ: This panel enables the user to select a specific area on the map by drawing a rectangle.

- **Filtering Panel** Ⓓ: This panel provides filtering operations over categorical attributes, allowing the user to perform faceted exploration.

- **Statistics Panel** Ⓔ: This panel presents statistics regarding the selected objects, such as univariate statistics (e.g., correlation, standard deviation) and bivariate statistics (e.g., Pearson correlation).

- **Analysis Panel** Ⓕ: This panel allows the user to perform data analysis tasks by selecting a visualization type (e.g., bar chart, heatmap, pie), data attributes, and an aggregate function.

6.3.2 UI Features

Here, we outline the basic features provided by the RawVis interface:

Map-based Visual Exploration. The user is able to explore, analyze and compare data over different geographical areas using operations like panning and zooming. For example, they may navigate and compute statistics over two different cities. Also, the user can focus on a specific area (e.g., neighborhood) by drawing a rectangle over the map Ⓒ.

Faceted Exploration. Faceted exploration and analysis is supported by allowing the user to define multiple filters over the categorical attributes via the Filtering panel Ⓓ. For example, in Figure 6.2, the user has selected to analyze the taxi trips which have been paid using a credit card (Payment type is CRD).

Statistics Computations. The interface offers the user the ability to analyze the data, through the statistics panel Ⓔ. Particularly, the user can examine univariate (e.g., mean, variance, standard deviation) or bivariate statistics (e.g., the Pearson correlation, covariance) over the data attributes. In Figure 6.2, univariate statistics have been computed for the fare amount, and bivariate for the fare and tip amount.

Visual Analysis. The user is able to visually analyze the data by selecting the most suitable visualization type and metrics to accomplish their analysis Ⓕ. Particularly, the user can select one or more variables to analyze, as well as the visualization type and metric. For example, in Figure 6.2, the user has selected a bar char to visualize the average tip value w.r.t. the payment method (e.g., cash, credit card). In the second case, the user has selected a heatmap, to visualize the average taxi fare per passenger count and payment type.

6.4 User Study

In order to study the usability and the performance of our tool, we conducted a user study. We considered a use case from the travel industry, using a dataset that contains information of approximately 180K hotels in the US gathered from multiple travel agencies. Each hotel is described by several attributes such as name, address, price, etc. The larger part of the data is retrieved from the public API of FACTUAL[2].

6.4.1 Setup

In our study, 44 participants took part. The participants were computer science graduate students, researchers, and analysts from the industry who were contacted via email. At the beginning of the evaluation, each participant was introduced to the system by an instructor who provided a brief tutorial on the required features for the tasks. After the instructions, the participants familiarized themselves with the system.

During the evaluation, each participant performed two tasks, as described below. In each task, we asked the participants to answer 10 Likert scale questions with five response options ranging from "Strongly agree" to "Strongly disagree." Some questions were related to the efficiency of the tool, such as "I found the 'Pan & Zoom' interactions efficient, reporting small response times." Another set of questions focused on the usability of the tool, for example, "I found it easy to obtain the requested information using a single chart." Finally, at the end of the evaluation, we asked the participants to answer questions regarding their overall experience.

6.4.2 Tasks Specification

To define the user tasks, we aimed to include several user interactions required for each task. Therefore, during the task specification, we considered the following well-known user interactions used in information visualization [97]. Specifically, we employed the following set of interactions to design the tasks and assess the capabilities of our tool:

Explore: Explore interaction techniques enable users to gradually examine specific subsets of the visualized datasets. Exploration refers to the set of operations with which a user can visualize only a part of a very large dataset at a time, examining it to gain understanding and insight, and then move on to view other parts of the data. An example of exploration is panning on a map or a scatter diagram, where the user visualizes only the data points within the visible screen window.

Filter: Filter techniques enable users to change the set of data items being presented based on a range of filter expressions. These expressions are conditions on specific attribute values that data points must have. Examples of filter conditions include arithmetic conditions (e.g., show me the population older than 40 years old), date-specific conditions (show me average temperatures between July and August), or categorical conditions (show me life expectancy rate only in Germany).

Connect: Connect techniques are usually employed to show associations and relationships between data items that are either in the same or different represen-

[2] https://www.factual.com/

tations. For example, in graph visualizations, a user may choose to show or hide a subset of the edges connecting the nodes of the graph. In multiple representations (e.g., a map associated with a bar chart), the user selects a data region on the map, and the associated value in the bar chart is highlighted, denoting an implicit relationship between the two visual representations.

Encode: Encode techniques allow users to change the visual appearance (e.g., color, size, and shape) of the data points. Visual appearance is important as it can help users better understand the differences, relationships, and distributions of the data elements. For example, by encoding height information to a map using a color scale, users can better identify the height information (e.g., the height of a mountain) without altering the spatial arrangement of the map.

Reconfigure: Reconfigure techniques provide users with the ability to change the layout, i.e., the spatial arrangement or alignment of the data in the visualization area, and present different perspectives. Examples include changing the X-axis of a bar diagram, the ordering (ascending or descending) of values, or the layout of a graph diagram.

Abstract: Abstract techniques allow users to adjust the level of abstraction of a data visualization, from an overview down to the details of individual data points. Abstraction often follows a hierarchical interaction relationship in the visualization, where the top-level overview contains many in-between levels of visualized data. A popular abstraction method is the zoom-in and zoom-out capability applied to data points visualized on a map. Through zooming, the scale of the map changes at a fixed set of abstraction levels, allowing users to see a bigger or smaller region of the map with all the contained data points.

6.4.3 Evaluation Scenario and Tasks

Consider the following example: You are a data analyst working for a consulting company that helps hotels advertise their business and offerings across booking platforms. Your company specializes in 4-star hotels. You wish to explore the data retrieved from booking platforms like Booking and Trivago, and analyze them based on hotel location, amenities, rating, and prices. The data is collected via available data APIs and stored in plain data files in raw formats (e.g., CSV) on your computer. You are requested to perform the following visual exploration tasks:

1. Gain an overview of which booking platforms have a high number of 4-star hotels. Navigate to the location of interest, filter out the 4-star hotels, and generate a chart showing the number of hotels per booking platform.

2. Inspect the platforms' coverage for different types of hotels and amenities and decide which one(s) cover most of your clients.

6.4.4 Tasks

In this section, we describe the tasks used in the user evaluation process. The tasks are designed to involve basic visual interactions and require users to test different functionalities of the tool.

Task 1. Navigate to Manhattan and zoom in to the maximum level. Select four-star hotels. From the chart panel, organize the hotels per booking platform and report the source with the largest number of hotels.

To solve this task, participants need to:

1. Pan and zoom to select the area they wish to explore (explore interaction).

2. Use the filtering panel to display only the four-star hotels (filter interaction).

3. Generate a graph (histogram, line, or area chart) that shows the COUNT of any measure, with the categorical attribute being the data source (connect interaction).

Overall, in this task, explore, filter, and connect interactions are involved.

Task 2. Navigate to Manhattan, and zoom in to the maximum level. Select four-star hotels. Find the booking platform(s) that cover all types of hotels in the visible neighborhood.

To solve this task, participants need to:

1. Pan and zoom to select the area they wish to explore (explore interaction).

2. Use the filtering panel to display only the four-star hotels (filter interaction).

3. Generate a heatmap with the categorical attributes being booking platform and hotel type, and the cells representing the aggregate number of hotels (COUNT) (connect interaction). The colors of the cells visually represent the relations between booking platforms, hotel types, and the number of hotels (encode interaction).

Overall, in this task, explore, filter, connect, and encode interactions are performed.

6.4.5 Results

In this section, we present the results of our user study.

6.4.5.1 Overall Experience

Figure 6.3 presents the responses regarding the overall experience of the users during the evaluation. We can see that 70% (30 out of 44) of the users strongly agree regarding the system efficiency, and 23% agree (83% combined). We can observe that, in all tasks, approximately 93% strongly agree and agree that the system is efficient.

Regarding the usability of the tool (Figure 6.3), approximately 95% of the users find it easy to use, rating it as strongly agree, agree, or borderline. Regarding task difficulty, 55% of the users found the tasks easy, while 38% of them encountered difficulties. These results can be explained by differences in the users' backgrounds.

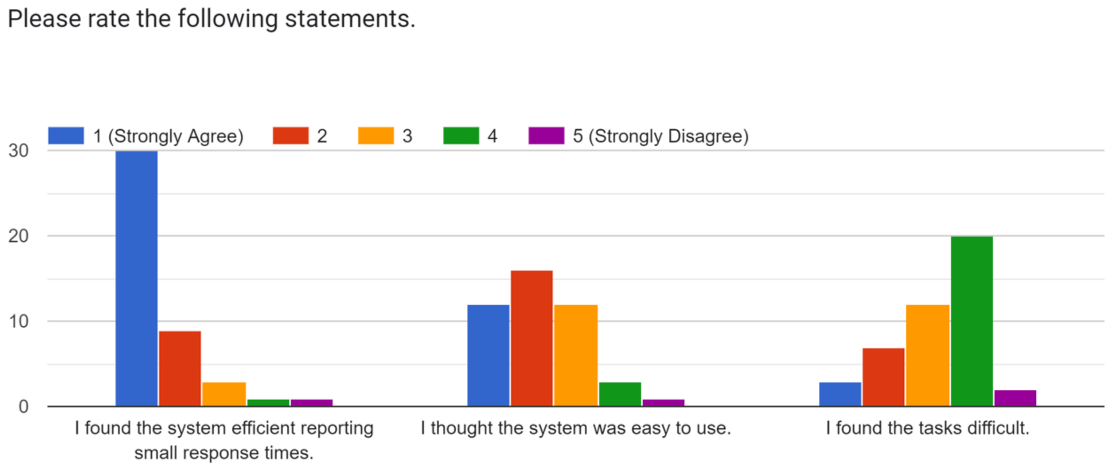


Figure 6.3: Overall User Experience

6.4.5.2 Features Usability

Figure 6.4 shows that the filter and chart generation features were found easy to use by the users, with 75% strongly agreeing or agreeing. The results are different for the feedback regarding extracting information from a single chart (Fig. 6.5, which proved more difficult for the users. This can be explained by the fact that Task 2 requires the use of a single heatmap chart, and participants have to inspect color variations in order to answer. This turned out to be challenging both due to the increased complexity of this chart and to the specific color scheme used in our implementation.

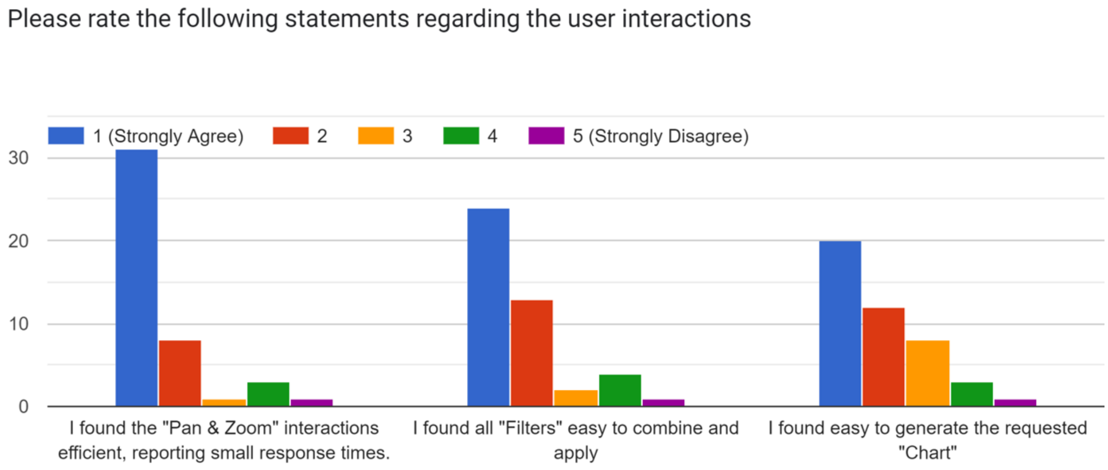


Figure 6.4: Task 1 Feedback

6.4.5.3 Time to Complete the Tasks

Figure 6.6 shows the approximate time users spent to accomplish each task. We can observe that for Task 1 and 2, more than 50% of the users spent less than 2 minutes,

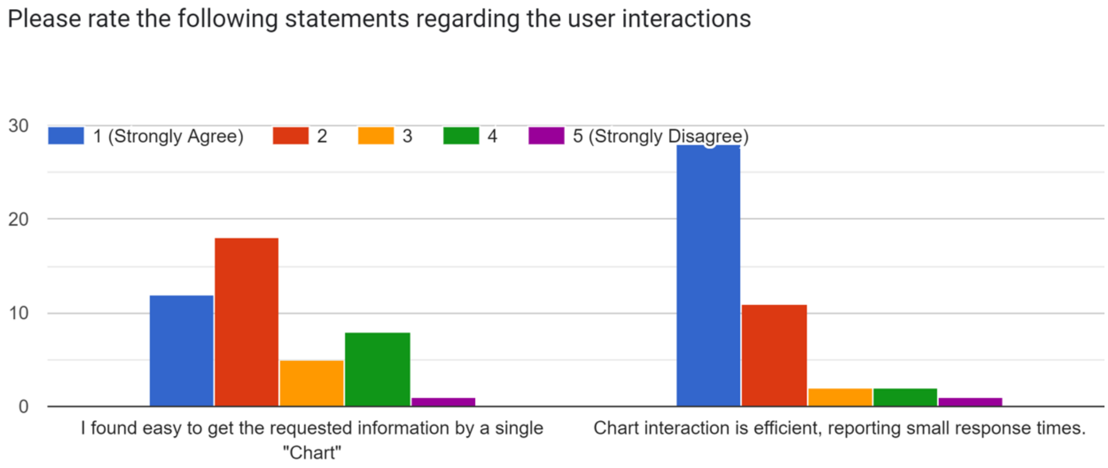


Figure 6.5: Task 2 Feedback

and about 15-20% spent less than 1 minute. In the case of Task 3, approximately 30% of the users spent about 3 minutes. The reported times depict the complexity of each task, with the first one being the easiest. Considering the tasks' requirements and difficulty, we believe that the reported times can be considered reasonable. Also, from Figure 6.6, we can observe that all users completed all the tasks, demonstrating their confidence in using the tool. Considering the time spent by users to accomplish the tasks and the high correctness of the responses provided, we can claim that the tool effectively assists users in accomplishing different tasks in various scenarios.

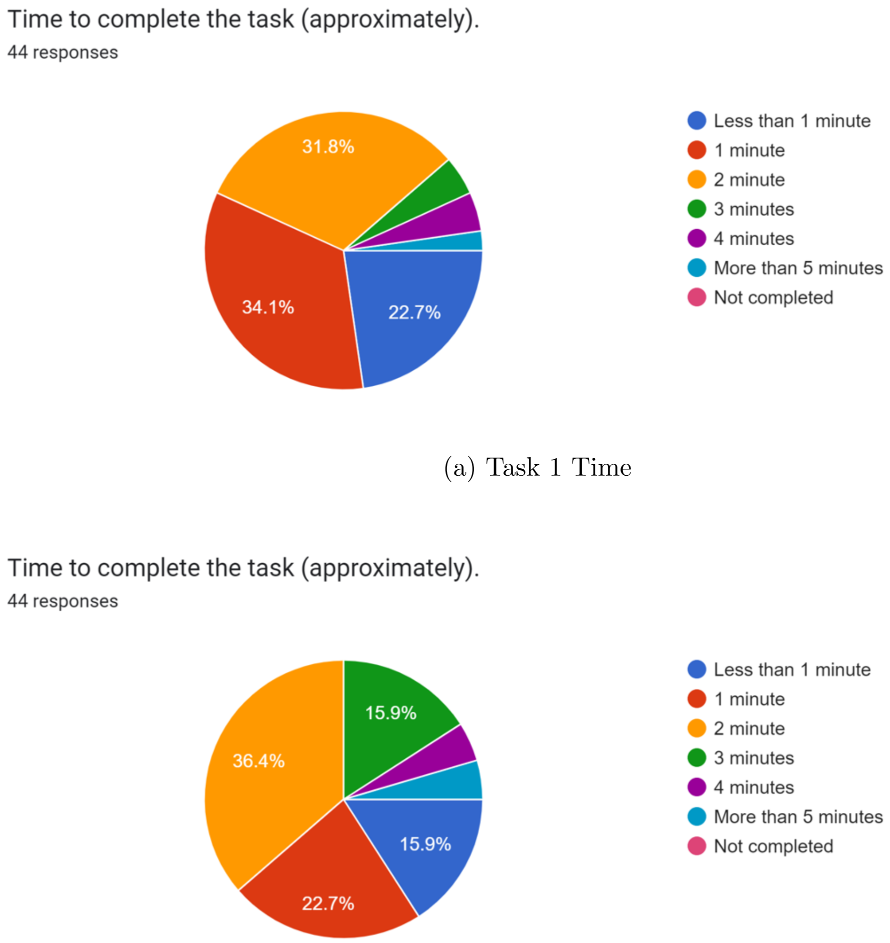
6.4.5.4 Results Summary

Here we outline the basic outcomes of our user study:

- The tool is highly efficient, with almost all users (more than 90%) agreeing that the tool has low response time in all actions.
- In each task, more than 70% of the users selected the correct answers.
- Considering the high correctness of answers and the time spent by users to accomplish the tasks, we can claim that the tool effectively assists users in accomplishing different tasks in various scenarios.
- The users showed confidence in using the tool, with all users completing all the tasks.
- There are some issues that need to be improved in the user interface.

6.5 Summary

This chapter introduced RawVis, a system for efficient, interactive exploration of large raw data files. We detailed the various aspects of RawVis's interface, exploring how it leverages techniques discussed in previous chapters to offer a powerful



(a) Task 1 Time

(b) Task 2 Time

Figure 6.6: Time taken for the tasks.

platform for the visual exploration of raw data, eliminating the necessity of relying on a traditional database management system.

Further, we presented and discussed the results of a user study that we conducted to examine the usability and performance of RawVis.

In conclusion, RawVis demonstrates the potential of the techniques discussed in this work, offering a substantial contribution to data exploration and visualization.

Chapter 7

Conclusions and Future Work

This work was driven by the increasing need for efficient and effective in-situ visual data analysis across a range of domains. It sought to address the challenges associated with in-situ processing and interactive visual analysis of large, raw data files. The ultimate aim was to develop new techniques and tools that can support data scientists in their work.

Central to this research was the premise that user interactions play a significant role in guiding the visual exploration and data analysis process. As such, a key aim was to incorporate these interactions into the proposed methods to achieve more efficient visual analysis.

7.1 Research Contributions

The research conducted in this work has resulted in several significant contributions, which are organized in accordance with the research objectives outlined in 1.3.

- **Minimize Data-to-Analysis Time:** User-driven initialization strategies were proposed to minimize the time required to start the analysis while ensuring efficient query evaluation. These strategies guided the construction of the VALINOR and VETI indexing schemes based on the initial user query. Instead of constructing fully detailed indexes, the indexes are built on-the-fly with enhanced granularity in the areas that are most relevant to the user's exploration, resulting in shorter construction time.

- **Efficient Evaluation of User Operations:** This work introduced a set of visual operations for 2D exploration, which were mapped to query operators over the underlying VALINOR and VETI indexing schemes. Moreover, it formulated exploratory and analytical operations over categorical attributes, thereby enabling efficient evaluation of user operations directly over raw files during in-situ exploration.

- **Minimizing I/O Operations:** The design of the indexing schemes, along with the efficient use of metadata, allowed for reduced I/O operations and faster user response times. These measures ensured efficient raw file parsing and reduced access to the file.

- **Optimizing Memory Usage:** A resource-aware approach to index initialization was implemented, which was formulated as an optimization problem.

Two approximation algorithms were provided to solve it, allowing for the optimal subset of data to be indexed and the index to be optimized within a predefined memory size.

- **Adapting to User Interaction:** Interaction-based adaptation techniques were designed that progressively adjust the index structure and metadata based on user interactions. These methods allowed for efficient query evaluation over the index in exploration scenarios and improved query evaluation efficiency by using information inferred from user interactions and analysis tasks.
- **Additional Contributions:** This work resulted in the RawVis open source visualization system, which implements the proposed methods. This system enables users to perform visual exploration and analytical operations over large raw datasets with low response times, even on commodity hardware, making it suitable for various interactive applications.

Overall, these contributions extend our understanding of how to enable efficient in-situ visual exploration and analysis of raw data, providing practical tools and techniques for the data science community.

7.2 Future Work

Despite the contributions of this work towards facilitating efficient in-situ visual exploration and analysis of raw data, there are various directions to further extend and enhance these efforts:

- **Expanding User Analytic Operations and Visualizations:** The research detailed in this work centers around exploratory and analytic operations tailored for the visual analysis of raw data files. Yet, the analytical spectrum of end-users often extends to more complex and domain-specific operations. Such operations could include advanced statistical analyses, outlier detection, or even predictive modelling executed directly on the raw data.

  In order to broaden the applicability of our indexing approach, future work could aim to adapt and enhance the current indexing schemes to effectively support such advanced operations. This could involve enriching the tile-based structure of the index and its accompanying metadata with additional statistical and aggregate information. This expansion would not only facilitate the execution of more complex operations but could also substantially improve their performance, providing a more robust and versatile platform for in-situ visual exploration and analysis.
- **Incorporating Spatial Partitioning Schemes:** The tile-based 2D index used in this work, while not explicitly designed for spatial coordinates, could potentially be enhanced by integrating established spatial partitioning schemes. Examples of such schemes include Geohash or Uber's H3[1], which are particularly beneficial in map-based exploration scenarios. These schemes offer efficient spatial partitioning at multiple resolutions and are extensively used for

[1] `https://eng.uber.com/h3/`

visual aggregation, such as heatmap generation. Incorporating these schemes into the index structure could not only improve performance in specific visual exploration tasks, but also expand the index's versatility in supporting different data types and queries. The main challenges lie in adapting these multi-level partitioning schemes to effectively complement the tile-based index and devising ways to maximize their utility for in-situ visual exploration of raw data. Given their inherent multi-level partitioning with more fine-grained divisions at some levels, these characteristics could be effectively harnessed within our index's adaptive and flexible structure.

- **Handling Time Series and Spatio-Temporal Data:** The proposed indexing schemes and exploration techniques present intriguing potential when applied to time series and spatio-temporal data, which offer a distinct set of challenges. These types of data are characterized by the continuous and voluminous influx of data points. Commonly, such data is streamed and stored in specialized time-series databases. However, in certain instances, they might also be batch-stored in file formats like CSV or Parquet within cloud storage systems. The efficient visual analysis of time series and spatio-temporal data, therefore, calls for the design and implementation of innovative techniques and algorithms. An initial direction could be the adaptation and augmentation of the indexing schemes developed in this work, introducing additional layers of indexing or metadata designed to handle the temporal and spatial dimensions effectively. Additionally, the indexing structures should be suitably enhanced to support the analysis of time series data stored across multiple batch files, allowing for seamless in-situ visual exploration over time-evolving datasets.

- **Approximate Visualization:** In the context of visual exploratory analysis, users often value speed and interactivity over total accuracy. This dynamic is especially noticeable during in-situ visual analysis, where parsing the necessary data objects from raw data files for certain operations can be time-consuming. Future work could explore the development of approximate query answering techniques specifically tailored to visual exploratory analysis. These techniques could involve a variety of strategies, from random sampling to statistical approximation, aiming to balance faster response times with a quantifiable degree of accuracy. In order to provide such approximate results with error bounds, the tile-based structure of the index and the metadata stored within each tile could be utilized. In this way, users could receive fast, approximate results without requiring a full data file access, while also being aware of the trade-off they are making between speed and accuracy.

- **Incremental Visualization Techniques:** Complementing approximate techniques, incremental visualization methods could provide further improvement. Such methods would allow users to receive and interpret intermediate results while data are still being processed. This would offer users the ability to get early insights from their data, which could be progressively refined as more data are processed. Possible approaches for realizing this could include developing progressive querying mechanisms that visualize partial results based on the data processed so far, coupled with the presentation of appropriate error metrics that help users judge the reliability of the visualized data.

These directions, among others, provide ample opportunities to build upon the foundations laid in this work. Future research in these areas has the potential to further improve the way data scientists work with large, raw data files.

Bibliography

[1] MySQL: The CSV Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/csv-storage-engine.html.

[2] Oracle: External Table Enhancements in Oracle Database 12c Release 1. https://oracle-base.com/articles/12c/external-table-enhancements-12cr1.

[3] PostgreSQL: Foreign Data. https://www.postgresql.org/docs/current/ddl-foreign-data.html.

[4] SciPy: Open Source Scientific Tools for Python. http://www.scipy.org.

[5] Tableau: Limitations to Data and File Sizes with Jet-based Data Sources. https://kb.tableau.com/articles/Issue/limitations-to-data-and-file-sizes-with-jet-based-data-sources.

[6] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *European Conference on Computer Systems (EuroSys)*, 2013.

[7] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient Query Execution on Raw Data Files. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2012.

[8] K. Alexiou, D. Kossmann, and P. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *VLDB Endowment*, 6(14), 2013.

[9] G. Andrienko, N. Andrienko, S. Drucker, J.-D. Fekete, D. Fisher, S. Idreos, T. Kraska, G. Li, K.-L. Ma, J. D. Mackinlay, A. Oulasvirta, T. Schreck, H. Schmann, M. Stonebraker, D. Auber, N. Bikakis, P. K. Chrysanthis, G. Papastefanatos, and M. Sharaf. Big Data Visualization and Analytics: Future Research Challenges and Emerging Applications. In *Workshop on Big Data Visual Exploration and Analytics (BigVis 2020)*, 2020.

[10] M. Angelaccio, T. Catarci, and G. Santucci. Query by diagram: A fully visual query system. *J. Vis. Lang. Comput.*, 1(3), 1990.

[11] M. Angelini, G. Santucci, H. Schumann, and H. Schulz. A Review and Characterization of Progressive Visual Analytics. *Informatics*, 5(3):31, 2018.

[12] L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2016.

[13] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 1990.

[14] S. Berchtold, D. A. Keim, and H. Kriegel. The X-tree : An Index Structure for High-Dimensional Data. In *Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[15] N. Bikakis. Big Data Visualization Tools. In *Encyclopedia of Big Data Technologies.* 2019.

[16] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis. Towards Scalable Visual Exploration of Very Large Rdf Graphs. In *Extended Semantic Web Conf. (ESWC)*, 2015.

[17] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis. Graphvizdb: A Scalable Platform for Interactive Large Graph Visualization. In *IEEE Intl. Conf. on Data Engineering (ICDE)*, 2016.

[18] N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis. A Hierarchical Aggregation Framework for Efficient Multilevel Visual Exploration and Analysis. *Semantic Web Journal*, 2017.

[19] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2014.

[20] D. F. Carbon, C. Henze, and B. C. Nelson. Exploring the SDSS Data Set with Linked Scatter Plots. I. EMP, CEMP, and CV Stars. *The Astrophysical Journal Supplement Series*, 228(2), 2017.

[21] S. K. Card, G. G. Robertson, and J. D. Mackinlay. The information visualizer, an information workspace. In *Intl. Conf. on Human Factors in Computing Systems (CHI)*, 1991.

[22] L. Caruccio, V. Deufemia, and G. Polese. Visual data integration based on description logic reasoning. In *IDEAS*, 2014.

[23] S. Chang. Visual Languages: A Tutorial and Survey. *IEEE Software*, 4(1), 1987.

[24] Y. Cheng and F. Rusu. SCANRAW: a Database Meta-operator for Parallel In-situ Processing and Loading. *ACM Transactions on Database Systems (TODS)*, 40(3), 2015.

[25] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Intl. Conf. on Very Large Databases (VLDB)*, 1997.

[26] S. Dar, M. J. Franklin, B. THór Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In *Intl. Conf. on Very Large Databases (VLDB)*, 1996.

[27] C. A. de Lara Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, Low Memory, Real-time Visual Exploration of Big Data. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 23(1), 2017.

[28] M. Derthick, J. Kolojejchick, and S. F. Roth. An Interactive Visualization Environment for Data Exploration. In *ACM Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, 1997.

[29] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for Visual Data Exploratio. In *Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, 2003.

[30] H. Ehsan, M. A. Sharaf, and P. K. Chrysanthis. Muve: Efficient Multi-objective View Recommendation for Visual Data Exploration. In *IEEE Intl. Conf. on Data Engineering (ICDE)*, 2016.

[31] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. Vistrees: Fast Indexes for Interactive Data Exploration. In *Workshop on Human-In-the-Loop Data Analytics (HILD)*, 2016.

[32] J. Fekete, D. Fisher, A. Nandi, and M. Sedlmair. Progressive Data Analysis and Visualization (Dagstuhl Seminar 18411). *Dagstuhl Reports*, 8(10), 2018.

[33] D. Fisher, I. O. Popov, S. M. Drucker, and M. C. Schraefel. Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. In *Intl. Conf. on Human Factors in Computing Systems (CHI)*, 2012.

[34] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2), 1998.

[35] P. Godfrey, J. Gryz, and P. Lasek. Interactive Visualization of Large Data Sets. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(8), 2016.

[36] D. Gotz and Z. Wen. Behavior-driven Visualization Recommendation. In *Intl. Conference on Intelligent User Interfaces (IUI)*, 2009.

[37] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Intl. Conf. on Extending Database Technology (EDBT)*, 2010.

[38] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.*, 1(1), 1997.

[39] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *VLDB Endowment*, 5(6), 2012.

[40] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2006.

[41] J. Heer and M. Bostock. Declarative Language Design for Interactive Visualization. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 16(6), 2010.

[42] P. Holanda and S. Manegold. Progressive mergesort: Merging batches of appends into progressive indexes. In *Conf on Extending Database Technology (EDBT)*, 2021.

[43] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt. Progressive Indexes: Indexing for Interactive Data Analysis. *PVLDB*, 12(13), 2019.

[44] P. Holanda, M. Nerone, E. C. de Almeida, and S. Manegold. Cracking kd-tree: The first multidimensional adaptive indexing (position paper). In *In Proc. of the Intl. Conf. on Data Science, Technology and Applications (DATA)*, 2018.

[45] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here Are My Data Files. Here Are My Queries. Where Are My Results? In *Conf. on Innovative Data Systems Research (CIDR)*, 2011.

[46] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Conf. on Innovative Data Systems Research (CIDR)*, 2007.

[47] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2009.

[48] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *VLDB Endowment*, 4(9), 2011.

[49] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2015.

[50] M. Ivanova, M. L. Kersten, S. Manegold, and Y. Kargin. Data Vaults: Database Technology for Scientific File Repositories. *Computing in Science and Engineering*, 15(3), 2013.

[51] A. H. Jensen, F. Lauridsen, F. Zardbani, S. Idreos, and P. Karras. Revisiting multidimensional adaptive indexing [experiment & analysis]. In *Conf on Extending Database Technology (EDBT)*, 2021.

[52] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. VDDa: Automatic Visualization-driven Data Aggregation in Relational Databases. *Journal on Very Large Data Bases (VLDBJ)*, 2015.

[53] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main Memory Evaluation of Monitoring Queries Over Moving Objects. *Distributed and Parallel Databases*, 15(2), 2004.

[54] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive Data Exploration Using Semantic Windows. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2014.

[55] I. Kamel and C. Faloutsos. On Packing R-trees. In *Intl. Conf. on Information and Knowledge Management*, 1993.

[56] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *VLDB Endowment*, 9(12), 2016.

[57] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on Raw Data. *VLDB Endowment*, 7(12), 2014.

[58] A. Key, B. Howe, D. Perry, and C. R. Aragon. Vizdeck: Self-organizing Dashboards for Visual Analytics. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2012.

[59] H. Lenz and B. Thalheim. OLAP Databases and Aggregation Functions. In *SSDBM*, 2001.

[60] S. T. Leutenegger, J. M. Edgington, and M. A. López. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *IEEE Intl. Conf. on Data Engineering (ICDE)*, 1997.

[61] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 19:2456–2465, 2013.

[62] C. Liu, C. Wu, H. Shao, and X. Yuan. Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE Trans on Visualization & Computer Graphics*, 26(1), 2020.

[63] Z. Liu and J. Heer. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12), 2014.

[64] J. D. Mackinlay, P. Hanrahan, and C. Stolte. Show Me: Automatic Presentation for Visual Analysis. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 13(6), 2007.

[65] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications.* Springer, 2006.

[66] A. Massari, S. Pavani, L. Saladini, and P. K. Chrysanthis. Qbi: Query by icons. In *ACM SIGMOD Record*, volume 24, 1995.

[67] F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva. TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 24:1394–1407, 2017.

[68] K. Morton, M. Balazinska, D. Grossman, and J. D. Mackinlay. Support the Data Enthusiast: Challenges for Next-generation Data-analysis Systems. *VLDB Endowment*, 7(6), 2014.

[69] B. Mutlu, E. E. Veas, and C. Trattner. Vizrec: Recommending Personalized Visualizations. *ACM Transactions on Interactive Intelligent Systems (TIIS)*, 6(4), 2016.

[70] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *ACM Conf on Management of Data (SIGMOD)*, 2020.

[71] M. Nerone, P. Holanda, E. C. de Almeida, and S. Manegold. Multidimensional Adaptive and Progressive Indexes. In *IEEE Conf on Data Engineering (ICDE)*, 2021.

[72] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1), 1984.

[73] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through Raw Data Via Adaptive Partitioning and Indexing. *VLDB Endowment*, 10(10), 2017.

[74] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal*, 2019.

[75] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *Conf on Extending Database Technology (EDBT)*, 2018.

[76] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2015.

[77] X. Qin, Y. Luo, N. Tang, and G. Li. Making data visualization more efficient and effective: a survey. *Journal on Very Large Data Bases (VLDBJ)*, 2020.

[78] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, 2018.

[79] P. Rahman, L. Jiang, and A. Nandi. Evaluating Interactive Data Systems. *VLDB J.*, 29(1), 2020.

[80] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld. I've Seen "Enough": Incrementally Improving Visualizations to Support Rapid Decision Making. *VLDB Endowment*, 10(11), 2017.

[81] S. Richter, J. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *Journal on Very Large Data Bases (VLDBJ)*, 23(3), 2014.

[82] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *VLDB Endowment*, 7(2), 2013.

[83] S. C. Seow. *Designing and Engineering Time: The Psychology of Time Perception in Software.* Addison-Wesley Professional, 2008.

[84] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Comput. Surv.*, 16(3), 1984.

[85] D. Sidlauskas and C. S. Jensen. Spatial Joins in Main Memory: Implementation Matters! *VLDB Endowment*, 8(1), 2014.

[86] D. Sidlauskas, S. Saltenis, C. W. Christiansen, J. M. Johansen, and D. Saulys. Trees or grids?: indexing moving objects in main memory. In *ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, 2009.

[87] W. Tao, X. Liu, Y. Wang, L. Battle, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive pan/zoom visualizations at scale. *Comput. Graph. Forum*, 38(3), 2019.

[88] F. Tauheed, L. Biveinis, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. Accelerating Range Queries for Brain Simulations. In *IEEE Intl. Conf. on Data Engineering (ICDE)*, 2012.

[89] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. SCOUT: Prefetching for Latent Feature Following Queries. *VLDB Endowment*, 5(11), 2012.

[90] Y. Tian, I. Alagiannis, E. Liarou, A. Ailamaki, P. Michiardi, and M. Vukolic. Dinodb: An Interactive-speed Query Engine for Ad-hoc Queries on Temporary Data. *IEEE Transactions on Big Data*, 2017.

[91] M. Vartak, S. Huang, T. Siddiqui, S. Madden, and A. G. Parameswaran. Towards Visualization Recommendation Systems. *SIGMOD Record*, 45(4), 2016.

[92] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *IEEE Trans on Visualization & Computer Graphics*, 23(1), 2017.

[93] C. Ware. *Visual Thinking: for Design.* Morgan Kaufmann, 2008.

[94] A. Wasay, X. Wei, N. Dayan, and S. Idreos. Data Canopy: Accelerating Exploratory Statistical Analysis. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2017.

[95] E. Wu, L. Battle, and S. R. Madden. The Case for Data Visualization Management Systems. *VLDB Endowment*, 7(10), 2014.

[96] S. Yesilmurat and V. Isler. Retrospective adaptive prefetching for interactive Web GIS applications. *GeoInformatica*, 16(3), 2012.

[97] J. S. Yi, Y. ah Kang, J. Stasko, and J. A. Jacko. Toward a deeper understanding of the role of interaction in information visualization. *IEEE transactions on visualization and computer graphics*, 13(6):1224–1231, 2007.

[98] E. T. Zacharatou, D. Sidlauskas, F. Tauheed, T. Heinis, and A. Ailamaki. Efficient Bundled Spatial Range Queries. In *ACM SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, 2019.

[99] F. Zardbani, P. Afshani, and P. Karras. Revisiting the theory and practice of database cracking. In *Conf on Extending Database Technology (EDBT)*, 2020.

[100] E. Zgraggen, A. Galakatos, A. Crotty, J. Fekete, and T. Kraska. How Progressive Visualizations Affect Exploratory Analysis. *IEEE Trans. Vis. Comput. Graph.*, 23(8), 2017.

[101] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity Join over Array Data. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2016.

[102] W. Zhao, F. Rusu, B. Dong, K. Wu, A. Y. Q. Ho, and P. Nugent. Distributed caching for processing raw arrays. In *Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, 2018.

[103] W. Zhao, F. Rusu, B. Dong, K. Wu, and P. Nugent. Incremental View Maintenance over Array Data. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2017.

[104] M. M. Zloof. Query-by-example: A data base language. *IBM systems Journal*, 16(4), 1977.

Research publications

Peer-reviewed journals:

- Maroulis, S., Bikakis, N., Papastefanatos, G., Vassiliadis, P., & Vassiliou, Y. (2022). Resource-aware adaptive indexing for in situ visual exploration and analytics. The VLDB Journal, 1-29.

- Bikakis, N., Maroulis, S., Papastefanatos, G., & Vassiliadis, P. (2021). In-situ visual exploration over big raw data. Information Systems, 95, 101616.

Peer-reviewed conference proceedings:

- Maroulis, S., Bikakis, N., Papastefanatos, G., Vassiliadis, P., & Vassiliou, Y. (2021, June). RawVis: A System for Efficient In-situ Visual Analytics. In Proceedings of the 2021 International Conference on Management of Data (pp. 2760-2764).

- Maroulis, S., Bikakis, N., Papastefanatos, G., Vassiliadis, P., & Vassiliou, Y. (2021). Adaptive Indexing for In-situ Visual Exploration and Analytics. In DOLAP (pp. 91-100).

- Bikakis, N., Maroulis, S., Papastefanatos, G., & Vassiliadis, P. (2018, September). RawVis: visual exploration over raw data. In European Conference on Advances in Databases and Information Systems (pp. 50-65). Springer, Cham.

Appendix A

Extended Greek Abstract

Εκτεταμένη περίληψη στα Ελληνικά

Προσαρμοστική ευρετηρίαση για διαδραστική οπτική εξερεύνηση και αναλυτική

A.1 Εισαγωγή

Η επέκταση των πρακτικών της ανοιχτής επιστήμης έχει οδηγήσει σε αυξημένη διαθεσιμότητα ανοιχτών συνόλων δεδομένων, μεγάλο μέρος από τα οποία παρέχονται με τη μορφή μεγάλων αρχείων δεδομένων(π.χ. CSV, JSON). Οι χρήστες που ενδιαφέρονται να εξερευνήσουν και να αναλύσουν τα δεδομένα αυτά, συχνά δεν έχουν εξειδικευμένες γνώσεις σε τεχνικές διαχείρισης δεδομένων, και διαθέτουν περιορισμένους υπολογιστικούς πόρους (π.χ. ένα laptop). Παράλληλα, συνήθως επιθυμούν την ανάλυση των δεδομένων αυτών με χρήση εύληπτων τεχνικών οπτικής εξερεύνησης κατευθείαν πάνω στα πρωτογενή αρχεία δεδομένων, αποφεύγοντας χρονοβόρες διαδικασίες για προετοιμασία των δεδομένων, όπως φόρτωση τους σε ένα Σύστημα Διαχείρισης Βάσεων Δεδομένων(ΣΔΒΔ), καθώς κι ευρετηρίασης τους σε αυτό.

Τα υπάρχοντα εργαλεία οπτικοποίησης είναι κατάλληλα για μικρά αρχεία που μπορούν να χωρέσουν στη μνήμη του συστήματος. Ωστόσο, αντιμετωπίζουν δυσκολίες με μεγαλύτερα αρχεία. Για την εξερεύνηση και ανάλυση αυτών των αρχείων, απαιτούνται χρονοβόρα βήματα προεπεξεργασίας όπως φόρτωση κι ευρετηρίαση, διαδικασίες οι οποίες προϋποθέτουν εμπειρία του χρήστη για την πραγματοποίηση τους. Αν και ορισμένες εμπορικές βάσεις δεδομένων υποστηρίζουν την εκτέλεση ερωτημάτων απ'ευθείας πάνω σε πρωτογενή αρχεία δεδομένων, χαρακτηρίζονται συνήθως από πολύ κακή απόδοση λόγω της ανάγκης για επανειλημμένη πρόσβαση στο αρχείο.

Πρόσφατα, έχουν προταθεί τεχνικές προσαρμοστικής ευρετηρίασης που στοχεύουν να αποφύγουν την αρχική κατασκευή ενός πλήρους ευρετηρίου, βελτιώνοντας αντίθετα το ευρετήριο και την ταξινόμηση των δεδομένων στη βάση κατά την εκτέλεση των ερωτημάτων. Ωστόσο, οι περισσότερες από αυτές τις τεχνικές απαιτούν ακόμη ένα στάδιο προεπεξεργασίας για τη φόρτωση των δεδομένων.

Παράλληλα, το in-situ πρότυπο που έχει αναδειχθεί τα τελευταία χρόνια, στοχεύει στην αποτίμηση ερωτημάτων απευθείας σε πρωτογενή αρχεία δεδομένων χωρίς την πλήρη φόρτωση και ευρετηρίαση τους σε ένα ΣΔΒΔ. Αυτό το πρότυπο δημιουργεί δομές κι ευρετήρια κατά τη διάρκεια της επερώτησης των δεδομένων από τον χρήστη και τα προσαρμόζει ανάλογα, με στόχο τη μείωση του κόστους πρόσβασης στα αρχεία και τη γρηγορότερη απάντηση στα ερωτήματα. Ωστόσο, οι περισσότερες υπάρχουσες εργασίες δεν επικεντρώνονται στις συγκεκριμένες ανάγκες για την υποστήριξη διαδραστικής οπτικής εξερεύνησης τέτοιων δεδομένων.

Αυτή η διατριβή επιχειρεί να αντιμετωπίσει αυτές τις προκλήσεις, επικεντρώνοντας στην αποτελεσματική in-situ οπτική ανάλυση δεδομένων. Οι στόχοι της είναι η αποτελεσματική αποτίμηση των ενεργειών εξερεύνησης και ανάλυσης του χρήστη πάνω σε πρωτογενή αρχεία δεδομένων, η δημιουργία ευρετηρίων με βάση την πρώτη ενέργεια του χρήστη με ελαχιστοποίηση του χρόνου αρχικοποίησης της ανάλυσης, η μείωση των λειτουργιών πρόσβασης στο αρχείο, η βελτιστοποίηση της χρήσης της μνήμης, και η προσαρμοστική ευρετηρίαση που προσαρμόζεται στα ερωτήματα του χρήστη. Ο τελικός στόχος είναι να προτείνει μια λύση που χειρίζεται αποτελεσματικά την in-situ οπτική εξερεύνηση και ανάλυση μεγάλων πρωτογενών αρχείων δεδομένων.

A.2 Επισκόπηση διατριβής

A.2.1 Μοντέλο Οπτικής Εξερεύνησης

Στο πλαίσιο της οπτικής αναλυτικής που στοχεύει η διατριβή, διατυπώνεται ένα μοντέλο οπτικής εξερεύνησης που παρέχει ένα σύνολο από διαθέσιμες αναλυτικές και εξερενητικές ενέργειες που μπορεί να εκτελέσει ένας χρήστης στα πρωτογενή αρχεία δεδομένων. Παράλληλα, παρέχεται μια σαφής αντιστοίχιση μεταξύ των οπτικών λειτουργιών και των διερευνητικών ερωτημάτων με στόχο τη συσχέτιση των αλληλεπιδράσεων των χρηστών με την πρόσβαση και την επεξεργασία των υποκείμενων δεδομένων.

Το μοντέλο οπτικής εξερεύνησης που παρουσιάζεται προϋποθέτει ένα αρχείο δεδομένων που περιέχει ένα σύνολο πολυδιάστατων αντικειμένων. Κάθε διάσταση αντιστοιχεί σε ένα πεδίο το οποίο μπορεί να είναι αριθμητικό, αλφαριθμητικό ή κατηγορικό. Επιπλέον, κάθε αντικείμενο συνδέεται με μια αριθμητική τιμή που υποδηλώνει τη θέση του εντός του αρχείου. Ιδιαίτερη έμφαση δίνεται στη 2διάστατη εξερεύνηση των δεδομένων. Ο χρήστης επιλέγει δύο αριθμητικά πεδία που μπορούν να αντιστοιχιστούν σε μια διάταξη δισδιάστατης οπτικοποίησης (π.χ. χαρτογραφική αναπαράσταση, διάγραμμα διασποράς). Οι χρήστες μπορούν να οπτικοποιήσουν μια ορθογώνια περιοχή, στην οποία απεικονίζεται το σύνολο των αντικειμένων του αρχείου των οποίων οι τιμές πέφτουν εντός του εύρους της περιοχής. Κάθε αντικείμενο της οπτικής αυτής αναπράστασης μπορεί επίσης να συσχετιστεί με οπτικά χαρακτηριστικά που αντιπροσωπεύουν τιμές από άλλα πεδία, πλην των δυο που αντιστοιχούν στους άξονες της δισδιάστατης απεικόνισης. Οπτικές λειτουργίες, όπως προβολή, μετακίνηση, μεγέθυνση/σμίκρυνση (zoom-in/out), φιλτράρισμα, ομαδοποίσης και ανάλυση, διευκολύνουν την εξερεύνηση του χρήστη στο δισδιάστατο χώρο και την περεταίρω ανάλυση των δεδομένων.

Κάθε οπτική λειτουργία από αυτές αντιστοιχείται τελικά σε τελεστές πρόσβασης στα αρχεία δεδομένων. Αυτοί οι τελεστές αποτελούν τη βάση ενός ερωτήματος εξερεύνησης, που στοχεύει στην ανάκτηση δεδομένων σύμφωνα με τις οπτικές λειτουργίες του χρήστη, παρέχοντας μια βάση για οπτική εξερεύνηση και ανάλυση δεδομένων.

A.2.2 Ευρετηρίαση για Δισδιάστατη Οπτική Εξερεύνηση

Για να επιτευχθεί δισδιάστατη εξερεύνηση αρχείων πρωτογενών δεδομένων από τους χρήστες, στο πλαίσιο της διατριβής αυτής παρουσιάζεται ένα ευρετήριο, το VALINOR. Το VALINOR αποτελεί ένα ευρετήριο που αποθηκεύεται στη μνήμη και είναι σχεδιασμένο ειδικά για δισδιάστατη in-situ οπτική εξερεύνηση μεγάλων αρχείων δεδομένων. Βασισμένο στο μοντέλο οπτικής εξερεύνησης, το VALINOR αποτιμάει αποτελεσματικά δισδιάστατες ενέργειες του χρήστη, όπως προβολή (render), μετακίνηση (move) και μεγέθυνση (zoom-in), πάνω στα αρχεία δεδομένων.

Ένα χαρακτηριστικό του VALINOR είναι η ιεραρχική δομή του, βασισμένη σε πλακίδια, η οποία ομαδοποιεί τα αντικείμενα ενός αρχείου βάσει δύο αριθμητικών χαρακτηριστικών, όπως το γεωγραφικό πλάτος και το γεωγραφικό μήκος στην περίπτωση μιας χαρτογραφικης οπτικοποίησης. Ένα πλακίδιο είναι μια υποπεριοχή του ευκλείδιου χώρου που ορίζεται από δύο διαστήματα στα 2 πεδία που αποτελούν τους άξονες της δισδιάστασης απεικόνισης. Τα πλακίδια στο τέλος της ιεραρχίας, που δεν έχουν άλλα πλακίδια απογόνους, ονομάζονται πλακίδια-φύλλα και σε αυτά αποθηκεύονται τα αντικείμενα του αρχείου. Συγκεκριμένα, για κάθε αντικείμενο αποθηκεύεται μια πλειάδα που περιέχει τις τιμές των πεδίων του άξονα και τη θέση του αντικειμένου στο αρχείο. Επιπλέον, το VALINOR ενσωματώνει μεταδεδομένα σύνοψης σε κάθε πλακίδιο. Αυτές είναι αριθμητικές τιμές που υπολογίζονται από αλγεβρικές συναρτήσεις (π.χ. άθροισμα, ελάχιστο, μέγιστο) πάνω σε όλα τα αντικείμενα σε ένα συγκεκριμένο πλακίδιο. Όταν ένα ερώτημα απαιτεί υπολογισμό κάποιων στατιστικών τιμών, αυτά τα μεταδεδομένα μπορούν να αξιοποιηθούν χωρίς να απαιτείται η ανάκτηση των αντίστοιχων πεδίων από το αρχείο. Η ικανότητα ενός ερωτήματος να εκμεταλλευτεί τα μεταδεδομένα εξαρτάται από το εάν το επικαλυπτόμενο πλακίδιο περιέχεται πλήρως ή εν μέρει στη δισδιάστατη περιοχή που ορίζεται στο ερώτημα. Σε ένα εν μέρει περιεχόμενο πλακίδιο, πρέπει να διατρέξουμε τα αντικείμενα του για να βρούμε αυτά που περιλαμβάνονται στο δισδιάστατο παράθυρο του ερωτήματος, και στη συνέχεια να διαβάσουμε τις τιμές για άλλα πεδία αυτών των αντικειμένων από το αρχείο. Αντίθετα, σε ένα πλήρως περιέχομενο πλακίδιο, δεν χρειάζεται να πραγματοποιήσουμε αυτές τις λειτουργίες καθώς τα μεταδεδομένα του μπορούν να χρησιμοποιηθούν χωρίς να χρειάζεται περεταίρω πρόσβαση στο αρχείο για τα αντικείμενα του.

Κατά την αλληλεπίδραση με μεγάλα αρχεία δεδομένων, ένα σημαντικό ζήτημα είναι η διαδικασία δυναμικής δημιουργίας ενός ευρετηρίου κατόπιν του πρώτου ερωτήματος ενός χρήστη, αντί για τη δημιουργία ενός πλήρως λεπτομερούς ευρετηρίου εκ των προτέρων. Αυτή η προσέγγιση παρέχει μια ισορροπία μεταξύ ταχύτητας και βάθους ανάλυσης, διασφαλίζοντας ότι ο αρχικός χρόνος απόκρισης σε ένα ερώτημα είναι ελάχιστος. Για το σκοπό αυτό, αρχικά δημιουργείται μια λιγότερο λεπτομερής εκδοχή του ευρετηρίου. Αυτή η εκδοχή είναι γρήγορη στην κατασκευή, διατηρώντας τον χρόνο για την αποτίμηση της πρώτης ενέργειας του χρήστη χαμηλό, αλλά περιέχει αρκετές λεπτομέρειες για να είναι χρήσιμη για την αρχική εξερεύνηση των δεδομένων.

Κατά την αρχικοποίηση του VALINOR, στη βασική μέθοδο αρχικοποίησης που παρουσιάζουμε, τα αντικείμενα ομαδοποιούνται σε πλακίδια ίδιου μεγέθους χωρίς να λαμβάνεται υπόψη η τοποθεσία του αρχικού ερωτήματος του χρήστη. Ωστόσο, σε ένα σενάριο εξερεύνησης, τα επόμενα ερωτήματα του χρήστη είναι πολύ πιθανό να επικαλύπτονται με πλακίδια κοντά στο αρχικό ερώτημα. Για να εκμεταλλευτούμε αυτήν την τοπικότητα, προτείνουμε μια μέθοδο αρχικοποίησης με βάση το πρώτο ερώτημα του χρήστη, όπου η αρχική δομή των πλακιδίων δημιουθργίεται με μια πιο λεπτομερή διαμόρφωση πλακιδίων στην περιοχή γύρω από το αρχικό ερώτημα. Η ιδέα είναι να αυξήσουμε τον αριθμό των πλακιδίων κοντά στο πρώτο ερώτημα, το οποίο αυξάνει την πιθανότητα επόμενα ερωτήματα του χρήστη να επικαλύπτονται με πλήρως περιέχομενα πλακίδια και μειώνει τα υπολογιστικά κόστη, καθώς και το κόστος ανάγνωσης από το αρχείο.

Συγκεκριμένα, η διαδικασία αρχικοποίησης ξεκινά με την κατασκευή ενός αρχικού συνόλου ισομεγέθων πλακιδίων χρησιμοποιώντας τη βασική μέθοδο αρχικοποίησης με πλακίδια ίσου μεγέθους. Στη συνέχεια, για κάθε ένα από τα αρχικά αυτά πλακίδια υπολογίζεται ένας συντελεστής διαίρεσης που καθορίζει τον αριθμό των ισομεγέθων υπο-πλακιδίων στα οποία θα διαιρεθεί το πλακίδιο. Ο συντελεστής αυτός υπολογίζεται βάσει της πιθανότητας που θα περιέχει ένα επόμενο ερώτημα το πλακίδιο. Στην πραγματικότητα, αυτή η τεχνική βασίζεται στην παραδοχή ότι τα επόμενα ερωτήματα του χρήστη θα είναι κοντά στο αρχικό ερώτημα, το οποίο είναι συχνά αληθές σε σενάρια όπως η αναλυτική επεξεργασία δεδομένων, όπου οι χρήστες συχνά εξετάζουν γειτονικές περιοχές.

Μετά την αρχικοποίηση του ευρετηρίου, τα επόμενα ερωτήματα του χρήστη μπορούν να το εκμεταλλευτούν για τη γρηγορότερη εκτέλεση τους. Η διαδικασία βασίζεται στη συνεχή ενημέρωση και βελτίωση του ευρετηρίου ως απάντηση στις αλληλεπιδράσεις των χρηστών. Αυτό επιτρέπει στο σύστημα να παρέχει πιο λεπτομερή αποτελέσματα καθώς γίνονται περισσότερα ερωτήματα. Η μέθοδος προσαρμογής του VALINOR, προσαρμόζει τη δομή των πλακιδίων με το να τα διαιρεί σε μικρότερα. Με αυτόν τον τρόπο, μειώνει τον αριθμό των αντικειμένων που χρειάζεται να ανακτηθούν από το αρχείο κατά τη εκτέλεση του ερωτήματος, καθώς τα μικρότερα πλακίδια είναι πιο πιθανό να περιέχονται πλήρως στο ερώτημα του χρήστη.

Κατά την εκτέλεση ενός ερωτήματος, εξετάζεται ποια από τα πλακίδια που επικαλύπτει το ερώτημα χρειάζεται να διαιρεθούν σε μικρότερα. Για τη διαμέριση ενός πλακιδίου σε μικρότερα υποπλακίδια, δοκιμάστηκαν διάφορες μέθοδοι. Για παράδειγμα, στη βασική μέθοδο που εξετάστηκε, ένα πλακίδιο που επικαλύπτεται με το ερώτημα του χρήστη μπορεί να χωριστεί σε 4 υποπλακίδια ίδιου μεγέθους (παρόμοια με την προσέγγιση που ακολουθείται στο Quad-tree). Το κύριο μειονέκτημα αυτής της μεθόδου είναι ότι σε πολλές περιπτώσεις όπου γίνεται η διαίρεση, δεν υπολογίζονται μεταδεδομένα για κανένα από τα δημιουργηθέντα υποπλακίδια. Αυτό συμβαίνει όταν τα πλακίδια που δημιουργούνται από τη διαίρεση δεν είναι πλήρως περιεχόμενα στο ερώτημα.

Για την αντιμετώπιση αυτού του προβλήματος, παρουσιάζουμε μια μέθοδο που αναδιοργανώνει το ευρετήριο με βάση το δισδιάστατο παράθυρο ερωτήματος του χρήστη. Η διαδικασία του χωρισμού περιλαμβάνει την κατασκευή υποπλακιδίων από ένα πλακίδιο βάσει της περιοχής επικάλυψης του με το ερώτημα. Με αυτή τη διαίρεση των πλακιδίων, η μέθοδος εξασφαλίζει ότι τουλάχιστον το παραγόμενο πλακίδιο που αντιστοιχεί στην τομή με το ερώτημα, θα είναι πλήρως περιεχόμενο μέσα στην περιοχή του ερωτήματος, και θα μπορέσει να αξιοποιήσει τις λειτουργίες εισόδου/εξόδου που εκτελούνται κατά τη διεργασία του ερωτήματος για τον υπολογισμό των μεταδεδομένων του. Με τον χωρισμό των πλακιδίων και τον υπολογισμό των μεταδεδομένων για τα υποπλακίδια, η μέθοδος μειώνει το κόστος πρόσβασης στο αρχείο και τον υπολογισμό των μεταδεδομένων για τα επόμενα ερωτήματα, βελτιώνοντας τελικά την εκτέλεση των ερωτημάτων.

Για την αντιμετώπιση σεναρίων εξερεύνησης με ιδιαίτερα περιορισμένη μνήμη, μέρος του VALINOR μπορεί να χρειαστεί να αποθηκευτεί στον δίσκο. Στο πλαίσιο αυτό, ορίζεται μια πολιτική που καθορίζει ποια μέρη του ευρετηρίου πρέπει να αφαιρεθούν από τη μνήμη και να γραφούν στον δίσκο. Η πολιτική που ακολουθείται είναι βασισμένη σε πλακίδια, και μεταφέρει τα αντικείμενα ενός πλακιδίου στο δίσκο. Για την επιλογή των πλακιδίων που θα μεταφερθούν στο δίσκο, ακολουθείται μια πολιτική με βάση την πιθανότητα να επικαλύπτει ένα επόμενο ερώτημα του χρήστη ένα πλακίδιο, και μεταφέρει στο δίσκο τα πλακίδια που βρίσκονται σε μεγαλύτερη απόσταση σε σχέση με το τελευταίο ερώτημα του χρήστη.

Κατά τη επεξεργασία ενός ερωτήματος, όταν ένα ερώτημα επικαλύπτεται με ένα πλακίδιο που έχει ήδη μεταφερθεί στο δίσκο, χρειάζεται να ανακτηθεί από αυτόν. Για να ανακτηθεί το απαιτούμενο πλακίδιο, απελευθερώνεται μνήμη με το να γραφεί ένα άλλο πλακίδιο στον δίσκο. Η πολιτική αποβολής στοχεύει στην ελαχιστοποίηση του συνολικού αναμενόμενου κόστους αποβολής ενώ εξασφαλίζει ότι τα αποβαλλόμενα πλακίδια περιέχουν τουλάχιστον έναν καθορισμένο αριθμό αντικειμένων. Από την άλλη, κατά τη φάση αρχικοποίησης του ευρετηρίου, η μεταφορά στο δίσκο πραγματοποιείται καθώς νέες εγγραφές διαβάζονται από το αρχείο δεδομένων και τοποθετούνται σε πλακίδια. Εάν η μνήμη γεμίσει, επιλέγονται τα πλακίδια που δεν έχουν προηγουμένως μεταφερθεί στο δίσκο, με βάση την απόσταση τους από το αρχικό ερώτημα του χρήστη. Τα αντικείμενα που βρίσκονται ήδη στη μνήμη γράφονται στον δίσκο. Αυτή η διαδικασία επιτρέπει την προσθήκη νέων αντικειμένων στο πλακίδιο μετά την αποβολή, πράγμα που σημαίνει ότι ορισμένα αντικείμενα μπορεί να βρίσκονται στη μνήμη ενώ άλλα αποθηκεύονται στον δίσκο.

Η ανακατασκευή είναι απαραίτητη κατά την επεξεργασία ερωτημάτων όταν ένα αποβαλλόμενο πλακίδιο επικαλύπτεται με ένα ερώτημα. Σε τέτοιες περιπτώσεις, τα αντικείμενα του πλακιδίου ανακτώνται από τον δίσκο και συνδυάζονται με τα αντικείμενα στη μνήμη για να ανακατασταθεί η πλήρης λίστα των αντικειμένων του πλακιδίου. Η αρχική σειρά εισαγωγής διατηρείται κατά τη διάρκεια αυτής της ανακατασκευής. Προκειμένου να ελαχιστοποιηθούν οι λειτουργίες εισόδου/εξόδου, δεν διαγράφονται αντικείμενα από τον δίσκο κατά τη διάρκεια της διαδικασίας ανάκτησης. Εάν ένα πλακίδιο χρειαστεί να αποβληθεί ξανά, αφαιρούνται μόνο τα αντικείμενα που δεν έχουν ήδη εγγραφεί στον δίσκο, απελευθερώνοντας μνήμη και γράφοντας τα στο δίσκο.

Αξιολογήσαμε πειραματικά το VALINOR χρησιμοποιώντας τόσο πραγματικά, όσο και συνθετικά σύνολα δεδομένων, σε σενάρια που περιλαμβάνουν την εξερεύνηση περιοχών των δεδομένων και τη μέτρηση των χρόνων εκτέλεσης και της κατανάλωσης μνήμης. Στη σύγκριση περιλαμβάνονται επίσης ανταγωνιστές όπως ένα παραδοσιακό σύστημα διαχείρισης βάσεων δεδομένων (DBMS), το σύστημα PostgresRaw [7], που αφορά στην εκτέλεση SQL ερωτημάτων κατευθείαν πάνω σε πρωτογενή αρχεία δεδομένων και μια υλοποίηση του R-tree στη μνήμη. Μετρώνται διάφορες μετρικές, όπως οι χρόνοι εκτέλεσης, η κατανάλωση μνήμης και οι λειτουργίες εισόδου/εξόδου. Τα αποτελέσματα δείχνουν ότι το VALINOR επιδεικνύει πολύ καλή απόδοση σε σύγκριση με άλλες λύσεις, ιδίως σε σενάρια με περιορισμένους πόρους μνήμης και για μεγάλα σύνολα δεδομένων.

Όσον αφορά την αρχικοποίηση ερωτημάτων, το VALINOR υπερτερεί σε σχέση με τις μεθόδους MySQL και R-tree για όλα τα σύνολα δεδομένων. Η MySQL απαιτεί σημαντικό χρόνο για την φόρτωση των δεδομένων στον δίσκο. Ο ελαφρώς υψηλότερος χρόνος αρχικοποίησης του VALINOR σε σύγκριση με τη PostgresRaw σε ορισμένες περιπτώσεις μπορεί να αποδοθεί σε μη βελτιστοποιημένη ανάγνωση του CSV και σε πιο αργές λειτουργίες εισόδου/εξόδου στην Java. Ωστόσο, το VALINOR εμφανίζει ταχύτερη εκτέλεση ερωτημάτων κατά την εξερεύνηση, όπου είναι περίπου 5-10 φορές πιο γρήγορο από τα υπάρχοντα συστήματα για τα περισσότερα σύνολα δεδομένων.

Όσον αφορά τον χρόνο εκτέλεσης ερωτημάτων, το VALINOR υπερτερεί σε σχέση με τους ανταγωνιστές. Η PostgresRaw απαιτεί παρόμοιο χρόνο για κάθε ερώτημα, καθώς χρειάζεται να εξετάσει όλα τα αντικείμενα στο σύνολο δεδομένων, ενώ η μέθοδος R-tree είναι σημαντικά πιο αργή σε σύγκριση με το VALINOR λόγω της έλλειψης μεταδεδομένων για αποδοτικό υπολογισμό συγκεντρωτικών στατιστικών. Ο συνολικός χρόνος που απαιτείται από το VALINOR για να εκτελέσει το πλήρες φορτίο ερωτημάτων είναι πολύ χαμηλότερος σε σύγκριση με άλλα συστήματα, καταδεικνύοντας την ανώτερη συνολική απόδοσή του.

Τα πειράματα κατανάλωσης μνήμης αποκαλύπτουν ότι, αν και το μέγεθος του VALINOR στη μνήμη αυξάνεται ελαφρώς καθώς επεξεργάζεται ερωτήματα λόγω του χωρισμού πλακιδίων και της αποθήκευσης μεταδεδομένων για πλήρως περιεχόμενα πλακίδια, έχει σημαντικά μικρότερες απαιτήσεις μνήμης σε σύγκριση με τους ανταγωνιστές.

Παράλληλα, αξιολογήσαμε τις μεθόδους αρχικοποίησης και προσαρμογής του. Συγκεκριμένα, η μέθοδος αρχικοποίησης με βάση το πρώτο ερώτημα του χρήστη βελτιώνει αποτελεσματικά το χρόνο απόκρισης στα πρώτα ερωτήματα του χρήστη. Αντίστοιχα, η μέθοδος προσαρμογής του ευρετηρίου με τη διαμέριση των πλακιδίων με βάση το ερώτημα, παρουσιάζει καλύτερη απόδοση προσαρμόζοντας τελικά το ευρετήριο καλύτερα σε σχέση με τις ενέργειες του χρήστη.

A.2.3 Ευρετηρίαση για Οπτική Εξερεύνηση σε κατηγορικά πεδία

Η δισδιάστατη οπτική εξερεύνηση δεδομένων, αν και σημαντική κι ευρέως διαδεδομένη (π.χ. χαρτογραφική οπτικοποίηση), αποτελεί ένα μέρος μόνο της εξερευνητικής ανάλυσης που μπορεί να θέλει να πραγματοποιήσει ένας χρήστης. Ένα άλλο είδος ανάλυσης που πραγματοποιείται συχνά, περιλαμβάνει τη δημιουργία γραφημάτων, όπως ένα γράφημα στηλών. Τέτοια γραφήματα συνήθως απαιτούν λειτουργίες ομαδοποίησης των δεδομένων ως προς κάποια κατηγορικά πεδία και τον υπολογισμό στατιστικών τιμών για κάθε ομάδα. Παράλληλα, τα κατηγορικά πεδία χρησιμοποιούνται συχνά και για το φιλτράρισμα των δεδομένων. Αυτές οι λειτουργίες έχουν βελτιστοποιηθεί σε παραδοσιακά συστήματα, όπως για παράδειγμα σε αποθήκες δεδομένων, αλλά απαιτούν τη φόρτωση των δεδομένων και τη δημιουργία κατάλληλων ευρετηρίων ή τον προϋπολογισμό όψεων των δεδομένων (π.χ. μοντέλο κύβου) για την γρηγορότερη αποτίμηση τους. Όμως, το in-situ σενάριο στο οποίο εστιάζει η διατριβή, αποσκοπεί στην αποφυγή του επιπλέον κόστους μετακίνησης και ευρετηρίασης δεδομένων σε ένα σύστημα διαχείρισης βάσεων δεδομένων και στη βελτίωση της απόδοσης μέσω της σταδιακής προσαρμογής ενός ευρετηρίου στη μνήμη κατά τη διάρκεια της εξερεύνησης των δεδομένων.

Το ευρετήριο VALINOR, αν και αποτελεσματικό για δισδιάστατα σενάρια εξερεύνησης, αντιμετωπίζει δυσκολίες στην εκτέλεση ενεργειών εξερεύνησης που περιλαμβάνουν κατηγορικά πεδία. Για το σκοπό αυτό, προτείνουμε ένα νέο σχήμα ευρετηρίασης, το ευρετήριο VETI, που σχεδιάστηκε για την αποτελεσματική εκτέλεση τέτοιων ενεργειών. Αυτό το ευρετήριο επεκτείνει το VALINOR με μια δενδρική δομή για το χειρισμό κατηγορικών τιμών, καθιστώντας το κατάλληλο για σενάρια εξερεύνησης που συνδυάζουν τη δισδιάστατη οπτική εξερεύνηση, μαζί με τη διανέργεια επιπλέον ανάλυσης με βάση τα κατηγορικά πεδία, όπως η δημιουργία γραφημάτων.

Η δομή CET που επεκτείνει το VALINOR, οργανώνει τα αντικείμενα με βάση τα κατηγορικά πεδία, με κάθε επίπεδο να αντιστοιχεί σε διαφορετικό πεδίο. Κάθε κόμβος συσχετίζεται με μια ακολουθία τιμών για τα πεδία αυτά που καθορίζεται από το μονοπάτι από τη ρίζα μέχρι εκείνο τον κόμβο. Τα αντικείμενα περιέχονται στα φύλλα του δέντρου. Τα φύλλα περιλαμβάνουν επίσης μεταδεδομένα σύνοψης - αριθμητικές τιμές που υπολογίζονται πάνω σε αριθμητικά πεδία των δεδομένων. Ο αριθμός των κόμβων σε ένα δέντρο CET εξαρτάται από την αντιστοίχιση των πεδίων στα επίπεδα του δέντρου και το αριθμό των διαφορετικών τιμών που αυτά μπορούν να πάρουν. Για το λόγο αυτό, τα πεδία με μικρότερο αριθμό τιμών τοποθετούνται πιο κοντά στη ρίζα, μειώνοντας τον αριθμό των κόμβων και των ακμών στο δέντρο.

Χρησιμοποιώντας τη δομή αυτή, παρουσιάζουμε το σχήμα ευρετηρίασης VETI, το οποίο συνδυάζει την δομή πλακιδίων του VALINOR με τη δομή δέντρου CET για την υποστήριξη αποτελεσματικών λειτουργιών τόσο στον δισδιάστατο χώρο όσο και σε κατηγορικά πεδία. Ένα πλακίδιο μπορεί να συσχετίζεται με ένα δέντρο CET που οργανώνει τα αντικείμενα με βάση ένα υποσύνολο κατηγορικών πεδίων. Διάφορα πλακίδια μπορεί να οργανώνουν αντικείμενα με βάση διαφορετικά σύνολα πεδίων, ενώ δεν είναι απαραίτητο όλα τα πλακίδια να έχουν ένα CET δέντρο.

Όπως και με το VALINOR, με το πρώτο ερώτημα του χρήστη, δημιουργείται αρχικά μια πιο ελαφριά έκδοση του VETI. Αυτή η έκδοση έχει σχεδιαστεί για να βελτιώσει την απόδοση των αρχικών αλληλεπιδράσεων του χρήστη χωρίς τη σημαντική αύξηση του χρόνου μέχρι την αρχική ανάλυση των δεδομένων. Η φάση αρχικοποίησης περιλαμβάνει τον καθορισμό των χαρακτηριστικών της δομής των πλακιδίων και των δέντρων τους, την ανάγνωση του αρχείου και την αντιστοίχιση των αντικειμένων στα πλακίδια και στα φύλλα των CET δέντρων. Η πολιτική αρχικοποίησης που χρησιμοποιείται στο VETI, λαμβάνει υπόψη το σημείο εισόδου της εξερεύνησης του χρήστη, το μέγεθος του παραθύρου του πρώτου ερωτήματος και τα κατηγορικά πεδία που περιλαμβάνονται στο πρώτο ερώτημα. Αυτή η πολιτική οδηγεί σε μια δομή πλακιδίων που είναι πιο λεπτομερής κοντά στο αρχικό ερώτημα και πιο αδρή όσο μεγαλώνει η απόσταση από αυτό.

Μετά την αρχικοποίηση του VETI, επόμενα ερωτήματα αποτιμώνται σε αυτό. Η εκτέλεση ενός ερωτήματος περιλαμβάνει αρκετά βήματα: εύρεση και προσαρμογή των επικαλυπτόμενων πλακιδίων και δέντρων, καθορισμός των αντικειμένων που απαιτούν πρόσβαση στο αρχείο, ανάγνωση των πεδίων των αντικειμένων από το αρχείο που δεν περιλαβμάνονται στα δέντρα, προσαρμογή των δέντρων, ενημέρωση των μεταδεδομένων και αποτίμηση του ερωτήματος. Το μοντέλο σταδιακής προσαρμογής του ευρετηρίου στο VETI αποσκοπεί στην προσαρμογή της δομής του ευρετηρίου στα ερωτήματα του χρήστη. Παράλληλα με τη διαίρεση των πλακιδίων, πραγματοποιείται και αναδιοργάνωση των αντικειμένων στα δέντρα τους, με την προσθήκη κατηγορικών πεδίων που ζητούνται στο τρέχον ερώτημα αλλά δεν υπάρχουν στα δέντρα.

Μια σημαντική πρόκληση όσον αφορά στην ευρετηρίαση κατηγορικών πεδίων στο πλαίσιο in-situ οπτικής εξερεύνησης, είναι οι αυξημένες απαιτήσεις σε μνήμη. Λαμβάνοντας υπόψιν τους περιορισμούς μνήμης που αντιμετωπίζουμε στο σενάριο που εξετάζουμε, ένα σημαντικό θέμα είναι ο προσδιορισμός των δέντρων CET που θα ανατεθούν σε κάθε πλακίδιο του ευρετηρίου. Ο στόχος είναι να προσδιορίσουμε τις δομές των δέντρων και να τα αναθέσουμε στα πλακίδια με βάση τη χρησιμότητά τους, λαμβάνοντας υπόψη τη διαθέσιμη μνήμη. Η χρησιμότητα (utility) του πλακιδίου αντιπροσωπεύει την πιθανότητα επικάλυψης του από μελλοντικό ερώτημα, ενώ ο βαθμός χρησιμότητας ενός κατηγορικού πεδίου αντιπροσωπεύει την πιθανότητα ζήτησης του συγκεκριμένου πεδίου από ένα μελλοντικό ερώτημα. Ο βαθμός χρησιμότητας ενός CET δέντρου καθορίζεται από το άθροισμα των επιμέρους βαθμών χρησιμότητας των πεδίων που περιλαμβάνονται σε αυτό. Ορίζουμε επίσης το βαθμό χρησιμότητας μιας ανάθεσης ενός δέντρου σε ένα πλακίδιο ως το συνδυαστικό βαθμό χρησιμότητας του πλακιδίου και του δέντρου. Το πρόβλημα αρχικοποίησης (SIN) περιλαμβάνει την ανάθεση δέντρων σε πλακίδια βάσει του βαθμού χρησιμότητάς τους.

Ο στόχος είναι να μεγιστοποιηθεί ο συνολικός βαθμός χρησιμότητας του ευρετηρίου, που είναι το άθροισμα των βαθμών χρησιμότητας όλων των αναθέσεων πλακιδίου-δέντρου, χωρίς το κόστος αρχικοποίησης να ξεπερνά τη διαθέσιμη μνήμη. Το κόστος αυτό, περιλαμβάνει τόσο τη μνήμη που απαιτεί η κατασκευή των πλακιδίων, όσο και αυτή για την κατασκευή των δέντρων. Η βέλτιστη λύση του προβλήματος SIN περιλαμβάνει την εξέταση των βαθμών χρησιμότητας όλων των πιθανών αναθέσεων δέντρων στα πλακίδια και την επιλογή του συνόλου αναθέσεων που μεγιστοποιεί τη συνολική χρησιμότητα, διασφαλίζοντας ότι το κόστος αρχικοποίησης του ευρετηρίου είναι μικρότερο από τον προϋπολογισμό της μνήμης. Αυτό απαιτεί την εξέταση ενός μεγάλου αριθμού αναθέσεων πλακιδίου-δέντρου, και μπορεί να αποδειχθεί ότι το πρόβλημα είναι NP-δύσκολο.

Για την αποτελεσματική λύση του προβλήματος, προτείνουμε δυο προσεγγιστικούς αλγόριθμους. Αρχικά, για τη μείωση του χώρου αναζήτησης, προσδιορίζεται ένα υποσύνολο από τα πιθανά δέντρα, που ονομάζονται υποψήφια δέντρα. Για τον προσδιορισμών των υποψήφιων δέντρων, τα κατηγορικά πεδία ταξινομούνται βάσει ενός βαθμού κέρδους που συνδυάζει το βαθμό χρησιμότητας του γνωρίσματος και το κόστος μνήμης του. Το σύνολο των υποψήφιων δέντρων αποτελείται από δέντρα που περιλαμβάνουν σταδιακά περισσότερα πεδία από την ταξινομημένη λίστα. Επιλέγοντας μόνο τα υποψήφια δέντρα, μειώνουμε σημαντικά τον αριθμό των αναθέσεων δέντρου που πρέπει να εξεταστούν. Παράλληλα, εκτιμούμε το κόστος μνήμης για κάθε ανάθεση πλακιδίου-δέντρου βάσει του μέγιστου αριθμού κόμβων που μπορεί να έχει ένα δέντρο. Αυτή η εκτίμηση βοηθά στη διαχείριση της διαθέσιμης μνήμης κατά τη διαδικασία επιλογής των αναθέσεων.

Ο πρώτος αλγόριθμος που προτείνεται, που ονομάζεται GRD, βρίσκει αναθέσεις πλακιδίου-δέντρου υπολογίζοντας το βαθμό χρησιμότητας για κάθε υποψήφια ανάθεση και επιλέγοντας τις αναθέσεις με το μεγαλύτερο βαθμό χρησιμότητας των οποίων το συνολικό εκτιμώμενο κόστος είναι μικρότερο από τον προϋπολογισμένο. Από την άλλη, ο 2ος προσεγγιστικός αλγόριθμος, που ονομάζεται αλγόριθμος ανάθεσης πλακιδίου-δέντρου βασισμένος σε ομαδοποίηση (binning), οργανώνει τα πλακίδια σε ομάδες και αναθέτει το ίδιο υποψήφιο δέντρο σε κάθε πλακίδιο μιας ομάδας. Ο αλγόριθμος BINN εξασφαλίζει μια πιο ισορροπημένη κατανομή της διαθέσιμης μνήμης και οδηγεί σε βελτιωμένη απόδοση σε σύγκριση με τον αλγόριθμο GRD. Στην προσέγγιση αυτή, τα πλακίδια ομαδοποιούνται με βάση διαστήματα πιθανοτήτων ή άλλους σχετικούς παράγοντες, και ξεκινώντας με τις ομάδες με μεγαλύτερο βαθμό χρησιμότητας, ανατίθεται ένα δέντρο με τα ίδια κατηγορικά πεδία σε όλα τα πλακίδια κάθε ομάδας, λαμβάνοντας υπόψιν τη διαθέσιμη μνήμη. Ο αλγόριθμος BINN επιτυγχάνει ταχύτερη επεξεργασία και απαιτεί λιγότερες αναγνώσεις αντικειμένων από το αρχείο σε σύγκριση με τον αλγόριθμο GRD.

Για την πειραματική αξιολόγηση του VETI στην αποτίμηση ερωτημάτων που περιλαμβάνουν κατηγορικά πεδία, εξετάσαμε την απόδοση του σε σχέση τόσο με το VALINOR, όσο και με ανταγωνιστές, όπως ένα παραδοσιακό ΣΔΒΔ και το σύστημα PostgresRaw για αφορά σε ένα σύστημα βασισμένο στην PostgreSQL για την αποδοτικότερη εκτέλεση ερωτημάτων απ'ευθείας σε πρωτογενή αρχεία δεδομένων. Το VETI επιδεικνύει χρόνους απόκρισης μικρότερους των 0,04 δευτερολέπτων στα περισσότερα ερωτήματα, ακόμα και κατά την επεξεργασία μεγάλων αρχείων έως και 45GB. Σε σύγκριση με τα πιο αποδοτικά ανταγωνιστικά συστήματα, το VETI επιτυγχάνει βελτιώσεις των χρόνων απόκρισης έως και 100 φορές και απαιτεί έως και δύο τάξεις μικρότερο αριθμό λειτουργιών Ε/Ε.

Παράλληλα, αξιολογήθηκαν οι αλγόριθμοι αρχικοποίησης που προτείνονται, με τον αλγόριθμο BINN να υπερτερεί γενικά από το GRD. Συγκεκριμένα, η αρχικοποίηση με τον αλγόριθμο BINN βελτιώνει την αποτίμηση των ερωτημάτων και είναι περισσότερο από 1,5 φορές γρηγορότερη σε σύγκριση με την αρχικοποίηση με το GRD. Ειδικότερα, ο BINN επιδεικνύει εξαιρετική απόδοση όταν ο χρήστης εξερευνά περιοχές που βρίσκονται πιο μακριά από το αρχικό ερώτημα ή όταν η διαθέσιμη μνήμη για το ευρετήριο είναι πολύ περιορισμένη. Συνολικά, η πειραματική αξιολόγηση υπογραμμίζει την υπερισχύουσα απόδοση του VETI, ειδικά όταν χρησιμοποιείται η αρχικοποίηση με χρήση του αλγορίθμου BINN.

A.2.4 Το σύστημα RawVis

Το μοντέλο οπτικής εξερεύνησης και οι τεχνικές ευρετηρίασης που παρουσιάζονται στο πλαίσιο της διατριβής, αποτελούν τη βάση του συστήματος RawVis, επιτρέποντας την αποτελεσματική ανάλυση των δεδομένων χωρίς την απαίτηση για χρονοβόρα προεπεξεργασία τους, όπως η φόρτωση και η ευρετηρίαση τους σε ένα ΣΔΒΔ. Το RawVis ακολουθεί μια αρχιτεκτονική πελάτη-διακομιστή και παρέχει τη δυνατότητα για οπτική εξερεύνηση δεδομένων απευθείας από τα πρωτογενή αρχεία, περιλαμβάνοντας μια πλούσια διεπαφή χρήστη και μια ευρεία γκάμα επιλογών για οπτικοποίηση κι εξερεύνηση. Το back-end του RawVis είναι υπεύθυνο για τη δημιουργία και διαχείριση των ευρετηρίων για την οπτική ανάλυση των πρωτογενών αρχείων. Παράλληλα, μετατρέπει τις αλληλεπιδράσεις του χρήστη σε λειτουργίες αποτίμησης των ερωτημάτων στα ευρετήρια και σε λειτουργίες Ε/Ε στα αρχεία δεδομένων. Με βάση τα ερωτήματα αυτά, τα ευρετήρια που δημιουργούνται προσαρμόζονται σταδιακά βελτιώνοντας την εκτέλεση των ερωτημάτων του χρήστη.

Η διεπαφή χρήστη επιτρέπει στο χρήστη να επιλέξει το αρχείο δεδομένων που επιθυμεί να εξερευνήσει, και παρουσιάζει μια χαρτογραφική αναπαράσταση των αντικειμένων του αρχείου για διαδραστική εξερεύνηση με λειτουργίες όπως μετακίνηση της περιοχής που εμφανίζεται (pan), μεγέθυνση ή σμίκρυνση (zoom-in/out), φιλτράρισμα, παραγωγή στατιστικών και ανάλυση των δεδομένων με δημιουργία γραφημάτων. Το εργαλείο υλοποιήθηκε χρησιμοποιώντας εργαλεία και βιβλιοθήκες ανοικτού κώδικα και είναι διαθέσιμο ως ανοιχτό λογισμικό.

Για την αξιολόγηση του συστήματος RawVis, πραγματοποιήσαμε μια μελέτη χρήστη που περιλάμβανε δύο εργασίες που σχετίζονταν με ένα σενάριο όπου οι συμμετέχοντες ανέλυαν δεδομένα από ξενοδοχεία. Τα αποτελέσματα της αξιολόγησης ήταν αρκετά ενθαρρυντικά, και η πλειονότητα θεώρησε το εργαλείο αποδοτικό και εύκολο στη χρήση, ενώ έγιναν προτάσεις για μελλοντική βελτίωση του, ειδικά όσον αφορά στη διεπαφή χρήστη.

A.3 Συνεισφορές της Διατριβής

Η παρούσα διατριβή είχε ως στόχο την αντιμετώπιση των προκλήσεων της επεξεργασίας ερωτημάτων in-situ και της διαδραστικής οπτικής ανάλυσης μεγάλων αρχείων δεδομένων. Οι συνεισφορές της παρούσας έρευνας περιλαμβάνουν:

- **Αποδοτική αποτίμηση των ενεργειών του χρήστη:** Αυτή η διατριβή πρότεινε τις δομές ευρετηρίασης VALINOR, CET και VETI για την υποστήριξη της διαδραστικής οπτικής εξερεύνησης απευθείας πάνω σε μεγάλα αρχεία πρωτογενών δεδομένων, χωρίς την απαίτηση για φόρτωση και ευρετηρίαση τους σε ένα παραδοσιακό ΣΔΒΔ.

- **Ελαχιστοποίηση αρχικοποίησης ανάλυσης:** Αναπτύχθηκαν μέθοδοι αρχικοποίησης των ευρετηρίων VALINOR και VETI βάσει του πρώτου ερωτήματος του χρήστη. Το ευρετήριο κατασκευάζεται δυναμικά, εξασφαλίζοντας μικρό χρόνο από την εισαγωγή των δεδομένων έως την ανάλυση, ακόμη και για πολύ μεγάλα σύνολα δεδομένων.

- **Ελαχιστοποίηση κόστους ανάγνωσης από το αρχείο:** Ο σχεδιασμός των σχημάτων ευρετηρίασης, μαζί με την αποδοτική χρήση των μεταδεδομένων, επέτρεψε τη μείωση των λειτουργιών I/O και των χρόνων απόκρισης του χρήστη. Αυτά τα μέτρα εξασφάλισαν αποτελεσματική ανάλυση αρχείων και μείωση της πρόσβασης στο αρχείο.

- **Βελτιστοποίηση Χρήσης Μνήμης:** Εφαρμόστηκε μια προσέγγιση που λαμβάνει υπόψη τους περιορισμούς της μνήμης για την κατασκευή του ευρετηρίου, η οποία διατυπώθηκε ως πρόβλημα βελτιστοποίησης. Παρέχονται δύο προσεγγιστικοί αλγόριθμοι για την επίλυσή του, επιτρέποντας τη βελτιστοποίηση του ευρετηρίου εντός ενός προκαθορισμένου μεγέθους μνήμης.

- **Προσαρμογή στις ενέργειες του χρήστη:** Σχεδιάστηκαν τεχνικές προσαρμογής βάσει των αλληλεπιδράσεων του χρήστη που προοδευτικά προσαρμόζουν τη δομή του ευρετηρίου και τα μεταδεδομένα.

- **Σύστημα RawVis:** Η παρούσα έρευνα οδήγησε στην ανάπτυξη του εργαλείου οπτικοποίησης RawVis, το οποίο υλοποιεί τις προτεινόμενες μεθόδους. Αυτό το σύστημα επιτρέπει στους χρήστες να πραγματοποιούν οπτική εξερεύνηση και αναλυτικές λειτουργίες σε μεγάλα αρχεία δεδομένων με χαμηλούς χρόνους απόκρισης..

Συνολικά, αυτές οι συνεισφορές επεκτείνουν την κατανόησή μας για το πως μπορούμε να παρέχουμε αποδοτικής in-situ οπτικής εξερεύνησης και ανάλυσης μεγάλων αρχείων δεδομένων, παρέχοντας πρακτικά εργαλεία και τεχνικές για την κοινότητα των επιστημόνων δεδομένων.

Appendix B

Glossary

Γλωσσάρι

Αγγλικός Όρος	Μετάφραση
adaptive indexing	προσαρμοστική ευρετηρίαση
aggregate function	συνάρτηση συσσώρευσης
aggregate statistics	συγκεντρωτικές στατιστικές
approximation algorithm	προσεγγιστικός αλγόριθμος
bar chart	γράφημα με ράβδους
big data	μεγάλα δεδομένα
binning	ομαδοποίηση
bivariate statistics	διμεταβλητές στατιστικές
categorical attributes	κατηγορικά πεδία
chart	γράφημα
complexity	πολυπλοκότητα
data exploration	εξερεύνηση δεδομένων
data object	αντικείμενο δεδομένων
dataset	σύνολο δεδομένων
Database Management System (DBMS)	Σύστημα Διαχείρισης Βάσεων Δεδομένων (ΣΔΒΔ)
dimension	διάσταση
distributed system	κατανεμημένο σύστημα
domain	πεδίο τιμών
exploratory data analysis	εξερευνητική ανάλυση δεδομένων
exploratory query	ερώτημα εξερεύνησης
file offset	θέση στο αρχείο
greedy algorithm	άπληστος αλγόριθμος
group-by operation	λειτουργία ομαδοποίησης
heatmap	θερμικός χάρτης
I/O operations	λειτουργίες εισόδου/εξόδου
in-situ query processing	επεξεργασία ερωτημάτων απευθείας στα αρχεία δεδομένων
incremental index adaptation	σταδιακή προσαρμογή ευρετηρίου
index	ευρετήριο

leaf	φύλλο
metadata	μεταδεδομένα
multi-dimensional indexing	πολυδιάστατη ευρετηρίαση
node	κόμβος
NP-hard	NP-δύσκολο
optimization problem	πρόβλημα βελτιστοποίησης
query	ερώτημα
raw data file	αρχείο πρωτογενών δεδομένων
scatter plot	γράφημα διασποράς
spatial index	χωρικό ευρετήριο
tile	πλακίδιο
tile splitting	διαίρεση πλακιδίων
user interface	διεπαφή χρήστη
user interaction	αλληλεπίδραση χρήστη
utility	χρησιμότητα
visual analysis	οπτική ανάλυση
visual exploration	οπτική εξερεύνηση