In-Situ Visual Exploration over Big Raw Data [1]

Nikos Bikakis[a], Stavros Maroulis[a], George Papastefanatos[a], Panos Vassiliadis[a]

[a] *University of Ioannina, Greece*

Abstract

Data exploration and visual analytics systems are of great importance in Open Science scenarios, where less tech-savvy researchers wish to access and visually explore big raw data files (e.g., json, csv) generated by scientific experiments using commodity hardware and without being overwhelmed in the tedious processes of data loading, indexing and query optimization. In this paper, we present our work for enabling efficient query processing on large raw data files for interactive visual exploration scenarios and analytics. We introduce a framework, named RawVis, built on top of a lightweight in-memory tile-based index, VALINOR, that is constructed on-the-fly given the first user query over a raw file and progressively adapted based on the user interaction. We evaluate the performance of a prototype implementation compared to three other alternatives and show that our method outperforms in terms of response time, disk accesses and memory consumption. Particularly during an exploration scenario, the proposed method in most cases is about 5-10× faster compared to existing solutions, and requires significantly less memory resources.

Keywords: Visual Analytics, Progressive & Adaptive Indexes, User-driven Incremental Processing, Interactive Indexing, RawVis, In-situ Query Processing, Big Data Visualization,

1. Introduction

Open science practices have provided, in recent years, a huge number of datasets, being shared in open access repositories. Many of these are produced, curated and consumed by scientists in the form of raw data, i.e., files in raw formats like .csv, .json, etc. These users usually have limited skills in complex data management and analysis as well as limited resources or commodity hardware for use. 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), avoiding the tedious tasks of data loading, preparation and indexing.

As an 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 an available datasets; e.g., Sloan

[1] To appear in Information Systems (IS), Elsevier, 2020

Digital Sky Survey (SDSS)[2], Palomar Transient Factory[3], Zwicky Transient Facility[4], Large Synoptic Survey Telescope[5], 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) [23]. Then, she 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/out* to explore a part of the region or a larger area, respectively. She 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. 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 user 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 [9], prohibitive for the interactive exploration purposes.

On the other hand, in-situ querying [48, 44, 9, 55, 67, 79, 68] is a recent trend, that aims at enabling the on-the-fly querying over large sets of raw data, by avoiding the loading and indexing overhead of traditional DBMS techniques. In such scenarios, large data files which *do not fit in main memory*, must be efficiently handled *on-the-fly* using *commodity hardware*. The techniques adopted in these scenarios, attempt to minimize the loading and I/O cost of querying by progressively building an index for the raw file in main memory.

Most of these works, however, study the generic in-situ querying problem without focusing on the specific needs for raw data visualization and exploration, and more specifically the need for *in-situ processing of a specific query class, that enables user operations in 2D visual exploration scenarios*; e.g., render data on a map, pan the visible area left or right, zoom or filter.

[2]www.sdss.org
[3]www.ptf.caltech.edu/iptf
[4]www.ptf.caltech.edu/ztf
[5]www.lsst.org
[6]For example, Tableau has limitations on the size of the data file that can be loaded for visualization [6].
[7]For example, Oracle [3], MySQL [1] and PostgresSQL [4] provide mechanism that enable SQL querying of csv files.

Although working in more than two dimensions or broader query classes is possible, both the 2.05-dimensional nature of the human eye [81] 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. Hence, the challenge in such scenarios is to achieve optimization of these specific operations, such that visual interaction with raw data is performed efficiently on very large input files using commodity hardware.

In this paper, we address the aforementioned challenge by developing a framework with specific, intentionally picked characteristics. Specifically, our proposed framework addresses the need for (1) in-situ, interactive visual exploration scenarios of 2D plots (e.g., maps or scatter plots); (2) over very large numbers of data points, residing in flat, external files on disk; (3) using commodity hardware (thus, alleviating the need for highly distributed computer infrastructure); and (4) without the overhead of a preprocessing step or the loading of data into a database. We consider a set of *fundamental* visual operations that are transformed to access operations to the raw data and propose query evaluation and optimization methods for improving their performance, i.e., the user response time during exploration.

To this end, we develop the RawVis framework, which is built on top of a lightweight main memory index, VALINOR (Visual AnaLysis Index On Raw data), constructed on-the-fly given the first user query. The index organizes information regarding the raw data objects into tiles in the 2D space; holds additional metadata in the tiles for enabling efficient analysis operations and overviews; and, adapts its structure based on the user interaction. In our extensive experiments we illustrate that the proposed framework *reports 5-10× faster response times*, during an exploration scenario, compared to existing systems. Next we provide an overview of the RawVis framework.

RawVis Framework Overview. In our working scenario (Figure 1), we consider that a *user* visually explores and analyzes data stored in a large *raw data file* in disk using a 2D visualization tool and technique (e.g., scatter plot, map). The visual interface makes use of our *RawVis framework* for accessing and querying the data file and our framework maps visual operations performed by the user to query operators on the data file.

As an example, let's assume that the user initially selects two attributes (A_x and A_y) of the file as the X and Y axis of the visualization. ❶ The *first time* the user requests to visualize or analyze the raw data, the raw file is parsed and a "simplified" initial version of the index is built, organizing the data objects into *tiles* based on their A_x and A_y values and storing metadata for the contents of each tile (*Index Initializer*). In parallel with the index construction, the first query is evaluated during the file parsing (*Query Evaluator*). ❷ All user's visual operations (e.g., pan, zoom, filter) *are transformed* to data access queries (*Visual Operation Translator*), which are then ❸ evaluated over the index structure (*Query Evaluator*). ❹ Following each query evaluation, the index structure is adapted accordingly, reorganizing the objects' grouping (*Index Adapter*). This process, which results in the incremental (i.e., progressively) adaptation of the index following the user interactions, constructs tile hierarchies, and recomputes and enriches metadata. 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. ❺ Finally, the results are returned to the user.

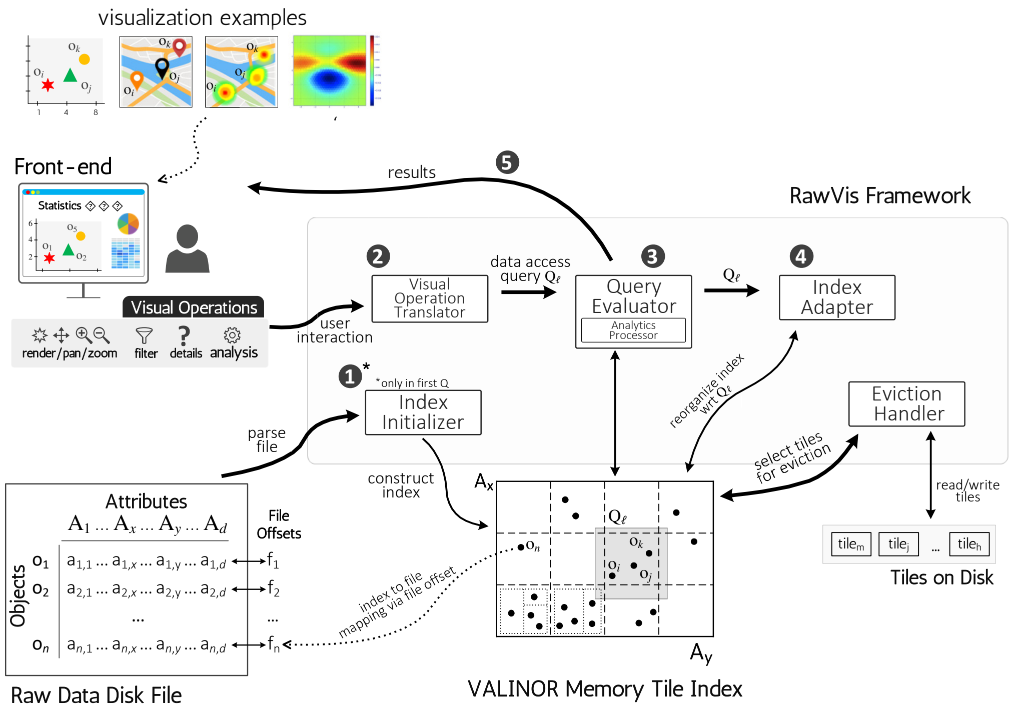


Figure 1: RawVis Framework Overview

Contributions. In this paper, we provide the following contributions:

- *We formulate visual user interactions over raw data as data-access operations*. Particularly, we propose a set of visual operations for 2D exploration and we map them to query operators over an underlying index.
- *We design a main-memory index in the context of in-situ visual exploration over large raw data*. The index exhibits a hierarchical tile-based structure for grouping objects based on X and Y dimensions, enriched with aggregated metadata (i.e., statistics about the contents of a tile) for enabling and speeding up analytics.
- *We introduce a user-driven initialization algorithm for building the index based on the first user query*. This methods is based on a locality based probabilistic approach that speeds up the user interaction at the initial stages of the exploration. Also, *we study the space complexity of the proposed index.*
- *We design a query-based adaptation technique that adjusts the index structure based on the user interaction. Also, we theoretically study the performance of the query evaluation w.r.t. to the proposed adaptation method.* Our adaptation algorithm splits the initial index structure into more fire-grained tiles following the user exploration and achieves increased performance especially for analytic tasks on the underlying data.
- We consider the case of memory constraints and *we implement an eviction mechanism for storing parts of the index in disk.*

– *We conduct an extensive experimental evaluation over two real-world and two synthetic datasets.* We evaluate the performance of our methods in terms of execution time, I/O operations, memory consumption and scalability. Moreover, we compare our framework with three competitors, i.e., MySQL, PostgresRaw [9], and R-tree, and we show that our framework outperforms competitors both in execution time and memory consumption. *Particularly, during an exploration scenario, our approach in most cases is about 5-10× faster compared to existing approaches.*

A preliminary version of this work outlines some basic concepts of our framework [20]. Here, we significantly extend [20] as follows. (1) In Sections 2, 3, 4, we formulate the basic concepts and provide the full details of the RawVis exploration model, the VALINOR index and the basic query answering mechanisms, respectively. (2) In Section 5, we present two completely novel extensions: (*i*) an extension to the initialization method of the index (Section 5.1); and (*ii*) a new method for the adaptation of the index based on the user query (Section 5.2). Both methods achieve greater performance in terms of response time compared to the initial baseline methods presented in [20]. (3) In Section 6, we design a new method for storing the index in disk. (4) In Section 7 we conduct extensive experiments with respect to [20] with two additional real datasets for evaluating both the baseline and the newly introduced techniques.

Outline. The paper is organized as follows. In Section 2 we present fundamental concepts of our framework, and in Section 3 we describe the proposed index. In Section 4 we present the basic operations over the index, that of index initialization and query evaluation. Then, Section 5 describes advanced techniques for the index initialization and adaptation, respectively, and Section 6 presents a disk-based implementation. Section 7 presents the experimental evaluation, and Section 8 the related work. Finally, Section 9 concludes the paper.

2. Basic Concepts

In this section we define the basic concepts of the RawVis framework. Table 1 presents the basic notations.

Raw Data File & Objects. We assume a *raw data file* $\mathcal{F}$ containing a set of *d-dimensional objects* 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.

Visual Exploration Model. Given a raw data file $\mathcal{F}$ containing a set of d-dimensional objects, the user arbitrarily selects[8] 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*.

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

Table 1: Common Notation

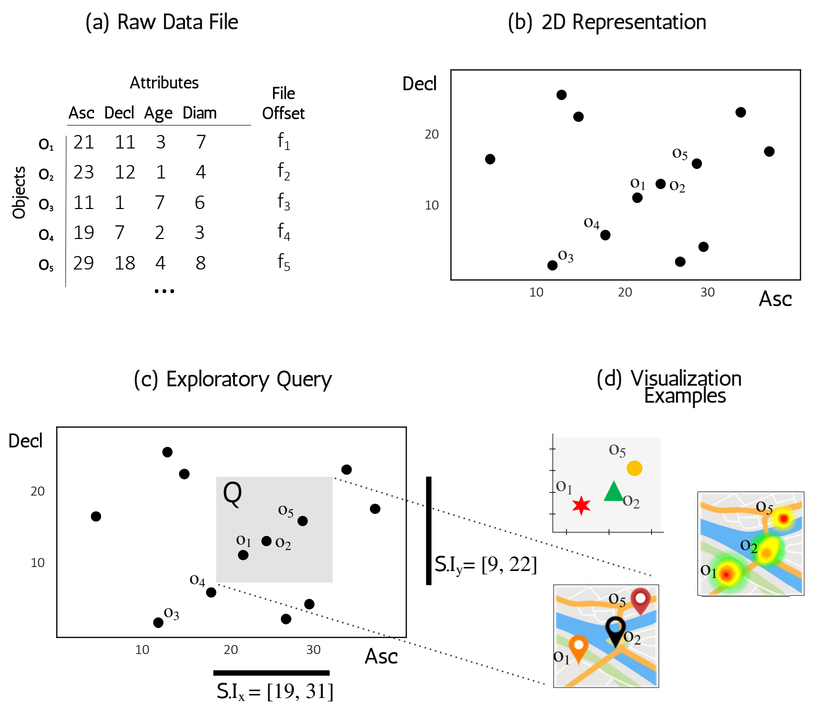
Symbol	Description
$\mathcal{F}$	Raw data file
O, o_i	Set of d-dimensional objects, an object
f_i	Position of o_i in the file $\mathcal{F}$
$\mathcal{A}$, A_j, $a_{i,j}$	List of attributes, the j$^{\text{th}}$ attribute of the list, the value of attribute A_j of the object o_i
A_x, A_y	Axis attributes
Φ, ϕ	2D visualized area, center of the visualized area
Q	Exploratory Query
S, F, D, N	Select, Filter, Details & Analysis part
O_{S}, O_{Q}	Objects selected from S, Objects resulted from O_{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 part
$(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$	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
IP, AP, 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

The user selects to visualize a rectangular area $\Phi = (I_x, I_y, O_\Phi, D_\Phi, N_\Phi)$, 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* $O_\Phi \subset O$, for which the values of their axis attributes fall within the ranges of that area. Each object $o_i \in O_\Phi$ 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 is associated with a set of visual annotations N_Φ calculated from applying a set of N aggregate functions to all objects O_Φ. The O_Φ, D_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 $\Phi' = (I'_x, I'_y, O'_\Phi, D'_\Phi, N'_\Phi)$. The following basic *visual operations* are considered:

– *render*: visualizes all objects contained in the visualized area. Formally:
$VO_{render} : \Phi(I_x, I_y, \varnothing, D_\Phi, N_\Phi) \rightarrow \Phi'(I_x, I_y, O_\Phi, D_\Phi, N_\Phi)$. 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:
$VO_{move} : \Phi(I_x, I_y, O_\Phi, D_\Phi, N_\Phi) \rightarrow \Phi'(I'_x, I'_y, O'_\Phi, D'_\Phi, N'_\Phi)$, where $I'_x = [x_1 + k_x, x_2 + k_x]$, $I'_y = [y_1 + k_y, y_2 + k_y]$



	Asc	Decl	Age	Diam	File Offset
o₁	21	11	3	7	f_1
o₂	23	12	1	4	f_2
o₃	11	1	7	6	f_3
o₄	19	7	2	3	f_4
o₅	29	18	4	8	f_5
		…			

Figure 2: (a) Raw Data File (b) 2D Data Representation (c) Exploratory Query (d) Visualization Examples

– *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: $VO_{zoom} : \Phi(I_x, I_y, O_\Phi, D_\Phi, N_\Phi) \rightarrow \Phi'(I'_x, I'_y, O'_\Phi, D'_\Phi, N'_\Phi)$, 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: $VO_{filter} : \Phi(I_x, I_y, O_\Phi, D_\Phi, N_\Phi) \rightarrow \Phi'(I_x, I_y, O'_\Phi, D'_\Phi, N'_\Phi)$, where $O'_\Phi \subset O_\Phi$.

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

– *analyze*: computes aggregate values for all objects included in Φ and visualizes them appropriately as annotations of the entire area. Formally: $VO_{analyze} : \Phi(I_x, I_y, O_\Phi, D_\Phi, \varnothing) \rightarrow \Phi'(I_x, I_y, O_\Phi, D_\Phi, N_\Phi)$.

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.

Example 1. [*Raw Data File & Objects*] In Figure 2(a) a sample of the raw data file is presented, 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* (dimensions). 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. Considering the object o_1, we have that $a_{1,1} = 21$, $a_{1,4} = 7$, etc. In analogy, for o_2 we have $a_{2,1} = 23$ and $a_{2,4} = 4$. Further, for each object o_i there is a *file pointer* f_i that corresponds to the offset of o_i from the beginning of the file.

Figure 2(b) presents a 2D representation of 12 file objects from the file, in which the attributes Asc and $Decl$ have been selected as the *axis attributes* A_X and A_Y, respectively. In this example, the attributes Age and $Diam$ are non-axis attributes. Assuming (not depicted in the figure) that we want to annotate each of the objects with the *details* of the Age of the star, each of the object would have an annotation with its Age (e.g., as a label or mouse-over tooltip). The area could also be annotated with the result of an analysis, that computes the *average Diam* of all the objects of the diagram. Finally, as shown in Figure 2(d), the objects included in the 2D visualized area are rendered using a scatter plot and a map. ■

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 d-dimensional objects O and the axis attributes A_x and A_y, an *exploratory query* Q over O is defined by the tuple $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$, where:

– The **Select part** 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 Select part is denoted as $\mathsf{S} = (I_x, I_y)$ with its intervals to be referred as $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. This part, selects the objects $O_{\mathsf{S}} \subseteq O$ for which both of their axis attributes have values within the respective intervals; i.e., their axis attributes' values are included in the 2D area (i.e., plane) specified by the intervals $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. The Select part is *mandatory* in a query Q, while the rest parts are *optional*.

– The **Filter part** F applies a set of (conjunction or disjunction) conditions c_i on the non-axis attributes. The Filter part is defined as $\mathsf{F} = \{c_1, c_2, ...c_k\}$. We denote the set of attributes involved in the conditions of F as F_A. Each condition c_i is a predicate involving an atomic unary or binary arithmetic operation over object attributes and constants. The Filter part is applied over the objects O_{S}, selecting the objects $O_Q \subseteq O_{\mathsf{S}}$ that satisfy the conditions in F. Note that, if the Filter part is not defined (i.e., $\mathsf{F} = \varnothing$), then $O_Q = O_{\mathsf{S}}$.

– The **Details part** D defines a set of non-axis attributes, for which the values of the objects that satisfy the filter will be returned by the query. Formally, $\mathsf{D} = \{A_1, A_2, ...A_k\}$, where $A_i \notin \{A_X, A_Y\}$, $\forall 1 \le i \le k$. The details part is applied on all filter surviving objects in O_Q.

–

The **Analysis part** N is a set of F_i algebraic aggregate functions [39], each of them applied over one or more numeric attributes of the objects O_Q. The Analysis part is defined as $\mathsf{N} = \{F_1(\mathcal{A}_1^{\mathsf{F}}), F_2(\mathcal{A}_2^{\mathsf{F}}), ...F_k(\mathcal{A}_k^{\mathsf{F}})\}$, where each function F_i takes as parameters a set of attributes belonging to $\mathcal{A}$ and returns a real numeric value. The result of each aggregate functions is a single numeric value for the entire visualized area, computed by applying it over the objects of O_Q. Also, as $\mathsf{N}_A = \bigcup_i^k \mathcal{A}_i^{\mathsf{F}}$ we denote the attributes that are used in all $F_i \in \mathsf{N}$. Further, $\mathcal{V}_{\mathsf{N}} = \{v_1, v_2, ...v_k\}$ is the list of values v_i calculated by F_i over the objects O_Q, with $v_i \in \mathbb{R}$. Note that, a large number of statistics (e.g., Pearson correlation, covariance) corresponds to the class of F_i. Specifically, more than 90% and 75% of the statistics supported by SciPy [5] and Wolfram [7], respectively, are defined as algebraic aggregate functions [82].

The semantics of query execution involves the evaluation of the four parts of the query in the following order: (1) *Select* part; (2) *Filter* part; (3) *Details*, *Analysis* part. Intuitively, the $Select$ and $Filter$ part apply restrictions (the equivalent of selection in relational algebra) to the entire space of objects, resulting in a set of qualifying objects O_Q. For each of these objects, we will visually present both the axis attributes as well as the attributes of the details part D (albeit with different treatments). Finally, we will apply the set of aggregate functions of the analysis part over all the objects of O_Q and, for each of these functions an aggregate numeric value that will also annotate the visualized area will be produced.

Query Result. The *result* $\mathcal{R}$ of an exploratory query Q over O is a set of tuples corresponding to the objects O_Q retrieved by the query as well as one or more numeric values for each function F_i applied on these objects. Each tuple contains the values of the axis attributes A_X and A_Y and the values of the attributes specified in D, denoted as $\mathcal{V}_{x,y,\mathsf{D}}$, as well as the numeric values $\mathcal{V}_{\mathsf{N}}$ resulted from computing the analytic part over O_Q. Formally, the result $\mathcal{R}$ consists of: (1) a set of tuples $\mathcal{V}_{x,y,\mathsf{D}} = \{\langle o_i : \alpha_{i,x}, \alpha_{i,y}, \alpha_{i,A_{\mathsf{D}_1}}, ...\alpha_{i,A_{\mathsf{D}_k}} \rangle, \forall o_i \in O_Q\}$, with $\{A_{\mathsf{D}_1}, ...A_{\mathsf{D}_k}\} = \mathsf{D}$; and (2) a set of numeric values $\mathcal{V}_{\mathsf{N}}$. Thus, the result is $\mathcal{R} = (\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$.

Example 2. [***Exploratory Query***] In Figure 2(c) an exploratory query Q is presented. The Select part 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 Select part are o_1, o_2, o_5. Assuming that the query has only a Select part, the result fetches only axis attribute values, i.e., $\mathcal{R} = (\langle o_1 : 21, 11\rangle, \langle o_2 : 23, 12\rangle, \langle o_5 : 29, 18\rangle)$. If we enrich the query with a Filter part $\mathsf{F} = \{Diam < 5 \text{ km}\}$, which applies a condition over the diameter attribute, i.e., $\mathsf{F}_A = \{Diam\}$, then the result will be $\mathcal{R} = (\langle o_1 : 21, 11\rangle, \langle o_5 : 29, 18\rangle)$, as the o_2 is omitted due to its 4km diameter. Furthermore, adding to the above query a Details part $\mathsf{D} = \{Age\}$, the result becomes $\mathcal{R} = (\langle o_1 : 21, 11, 3\rangle, \langle o_5 : 29, 18, 4\rangle)$. Finally, assume that the query defines an Analysis part $\mathsf{N} = \{corr(Age, Diam), Avg(Age)\}$, which calculates the *correlation* (i.e., Pearson correlation coefficient) between *Age* and *Diam*, and the *average Age*, i.e., $F_1 = corr$, $F_2 = Avg$, and $N_A = \{Age, Diam\}$. The two functions are computed only over the objects included in the query result; i.e., o_1 and o_5. So, the result is $\mathcal{R} = (\{\langle o_1 : 21, 11, 3\rangle, \langle o_5 : 29, 18, 4\rangle\}, \{0.996, 3.5\})$, where $v_1 = 0.996$ and $v_2 = 3.5$ are the *correlation* between Age and Diam, and the *average* Age of o_1 and o_5, respectively. ■

Table 2: 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, $\{A_1, A_2, ...A_k\}$	$\mathsf{S} = (I_x, I_y)$ $\mathsf{D} = \{A_1, A_2, ...A_k\}$
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 $I_x \times I_y$

Expressing Visual Operations as Exploratory Queries. Each visual operation of our model can be implemented by a data access operator of an exploratory query. Table 2 presents the correspondences for the six aforementioned visual operations.

The *Render* operation is implemented by the Select part of the query, setting the intervals I_x and I_y equal to the region of the visualized 2D area Φ. The *move* operation changes the current visualized area Φ shifting to a new one Φ'. It is again implemented by the Select part of a query with the new intervals of the shifted area Φ'. The $zoom\ in/out$ operations are also implemented by a Select part, having as interval parameters the new coordinates of the contained/containing visualized regions, respectively. Note that, $|I|$ denotes the length of the interval I.

The $filter$, *details* and *analyze* operations are implemented in a straightforward manner by including in the Select part the appropriate data access operator, i.e., the $filter$ operation is implemented by including a Filter part, and the *details* and *analyze* operations correspond to the Details and Analysis parts, respectively. Note that, as described in the previous section, multiple visual operations can be combined and implemented with one query, e.g., move a region while filtering the objects based on the value of a specific attribute.

3. The VALINOR Index

The VALINOR 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 progressively adjusts its structure to the user visual interactions. Each tile is constructed, during initialization, on range over values of the A_x and A_y axis attributes, by dividing the Euclidean space into initial tiles (see Sect 5.1 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.

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 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 $x_1 \leq x'_1, x_2 \geq x'_2, y_1 \leq y'_1$ and $y_2 \geq y'_2$. 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 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 th 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 [57]. 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.

VALINOR Index. 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 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.

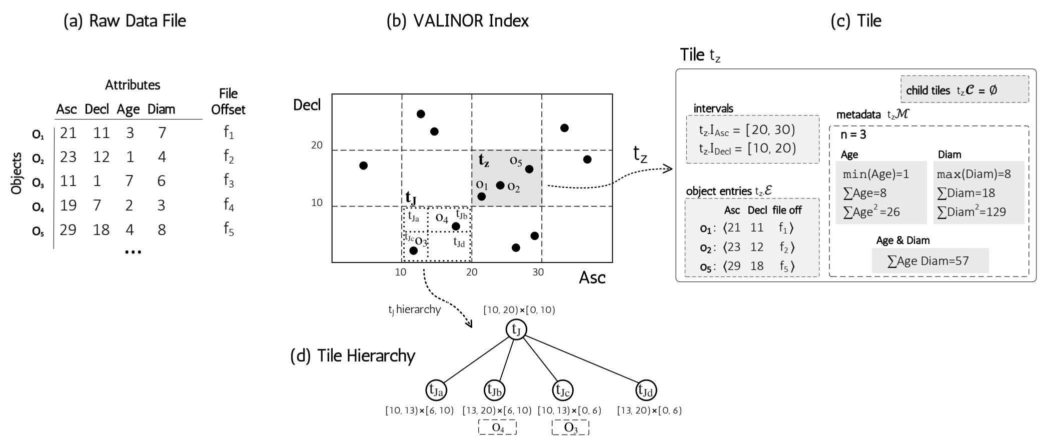


Figure 3: The VALINOR Index Overview

Tiles-Query Spatial Relations: Overlap, Fully/Partially-Contained. Considering the spatial relations between the Select part of the query and tiles included in the VALINOR, we define the following.

Given an exploratory query Q with S be the Select part and $\mathcal{T}$ be the tiles defined 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.

Example 3. [*VALINOR Index*] Figure 3(a) presents the input data file from Example 1, where the Asc and Decl have been selected as the two axis attributes for the analysis. 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 3(d).

Figure 3(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*. ■

Implementation Details & Practical Considerations. To make our implementation work, we have adopted several design choices and assumptions. We assume that data in a CSV file are organized in records separated by a new line delimiter (we assume that no headers exist in the file); all records are homogeneous, and thus have the same schema (same number of entries). Entries within a record are separated by a comma delimiter, followed by a new line character indicating the beginning of the new record in the file. The offset of the record is the location of its first character in the file, i.e., the hex value of the location of the next character after the new line delimiter. Each tile is linked with its records in the data file by maintaining a list with the offset (hex value) of the beginning of each record. When parsing the raw file, we reduce the tokenizing and parsing costs by parsing only the necessary attributes for a query and stopping the tokenizing once we find the last attribute in the row needed for the query or initialization. The user has minimal input to give, specifically, the delimiter of the csv file (e.g., comma or tab) as well as which are the axis attributes and a reasonable estimate for their ranges in order to avoid having to scan the raw file to determine them. Each object in a tile contains exactly two floats (x, y values) and a long value (offset). The index tiles are not of uniform size. When tiles are evicted to the disc, only their objects are actually written. We do not rely on Java serialization for this.

4. Query Processing over 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.

4.1. 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 $\mathsf{computeInitialTileSize}$ method to determine an initial tile size ℓ_{x_0}, ℓ_{y_0} (*line* 2). Then, using this initial size, the $\mathsf{constructInitialTiles}$ 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.

The initialization policy IP determines the initial tile sizes. 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) [50, 14, 78, 21]; or, (3) computed from the data in the raw file based on a binning technique that divides the data space into equal size tiles. The latter was introduced in [20] and is considered as the baseline method for the initialization of the index. In Section 5.1, 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

Algorithm 1. 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: $V$: the attribute values used in Analysis part computation

1  $V \leftarrow \varnothing$
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 Select part S and satisfies the Filter part 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 part 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 $V$ the values of $o_i$ for the attributes $\mathsf{N}_A$ referred in the Analysis part A
13 $\mathcal{V}_{\mathsf{N}} \leftarrow$ use the values of $V$ to compute the statistics of the Analysis part A
14 return $\mathcal{I}$, $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_{\mathsf{N}})$
```

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 part, and satisfies the conditions of the Filter part. Then, it reads the attribute values in the Details part, 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 part for each tile (*line* 13). Finally, the result of the first query and the initialized index are returned (*line* 14).

4.2. *Query Processing Overview*

The following process describes the evaluation of all subsequent queries. An overview of the query evaluation is presented in the Algorithm 2 and details for each operator are provided in following subsections. Algorithm 2 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* part of the query, and, (b) the values computed for each tile in the Analysis part.

First, the Select part is evaluated (*line* 1), using the evaluateSelectPart 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 Select part 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).

Algorithm 2. 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: O_S: objects selected from Select part; $\mathcal{T}_\mathsf{S}$: leaf tiles that overlapped with the Select part;
$\mathcal{T}_{\mathsf{S}_\mathcal{F}}$: 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 part; $\mathcal{V}_\mathsf{D}$: values of the attributes defined in the Details part;
$\mathcal{V}_{\mathsf{N}_A}$: values of the attributes required for the Analysis part 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 part

Parameters: AP: adaptation policy; MH: metadata handler

Output: $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_\mathsf{N})$: query result $\mathcal{R}$

```
1  $O_\mathsf{S}, \mathcal{T}_\mathsf{S} \leftarrow$ evaluateSelectPart ($\mathcal{I}$,S)
2  $\mathcal{T}_{\mathsf{S}_\mathcal{F}} \leftarrow$ getTilesRequireFileAccess ($\mathcal{T}_\mathsf{S}$, $Q$)
3  $\mathcal{T}'_{\mathsf{S}_\mathcal{F}} \leftarrow$ AP.adaptTiles ($\mathcal{T}_{\mathsf{S}_\mathcal{F}}$, $O_\mathsf{S}$)
4  **if** $\mathcal{T}_{\mathsf{S}_\mathcal{F}} \neq \varnothing$ **then**
5      $\mathcal{V}_{\mathsf{F}_A}, \mathcal{V}_\mathsf{D}, \mathcal{V}_{\mathsf{N}_A}, \leftarrow$ readFile ($\mathcal{T}'_{\mathsf{S}_\mathcal{F}}$, $O_\mathsf{S}$, $Q$, $\mathcal{F}$)
6  **if** $\mathcal{T}'_{\mathsf{S}_\mathcal{F}} \neq \mathcal{T}_{\mathsf{S}_\mathcal{F}}$ **then**
7      MH.updateMetadata ($\mathcal{T}'_{\mathsf{S}_\mathcal{F}}$, $Q$, $\mathcal{V}_{\mathcal{A}_\mathsf{F}}$, $\mathcal{V}_{\mathsf{N}_A}$ )
8  $O_Q \leftarrow$ evaluateFilterPart ($O_\mathsf{S}$, $\mathcal{V}_{\mathsf{F}_A}$)
9  $\mathcal{V}_{x,y,\mathsf{D}} \leftarrow$ construct the tuples by combining $O_Q$ and $\mathcal{V}_\mathsf{D}$
10 $\mathcal{V}_\mathsf{N} \leftarrow$ evaluateAnalysisPart ($O_Q$, N, $\mathcal{V}_{\mathsf{N}_A}$)
11 **return** $(\mathcal{V}_{x,y,\mathsf{D}}, \mathcal{V}_\mathsf{N})$
```

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 parts, 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) parts are evaluated.

Example 4. [***Query Processing***] In this example we assume the same exploratory query as the one used in Example 2. Particularly, the query Q has the following parts: (1) *Select part*: $\mathsf{S}.I_x$=[19°, 31°], $\mathsf{S}.I_y$=[9°, 22°]; (2) *Filter part*: $\mathsf{F} = \{Diam < 5$ km$\}$; and (3) *Analysis part*: $\mathsf{N} = \{corr(Age, Diam), Avg(Age)\}$. Further, we assume the index described in Example 3 and presented in Figure 3.

The query processing procedure is depicted in Figure 4. 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 Select part. 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 part defined in the query. Recall that, the metadata is computed and stored per tile.

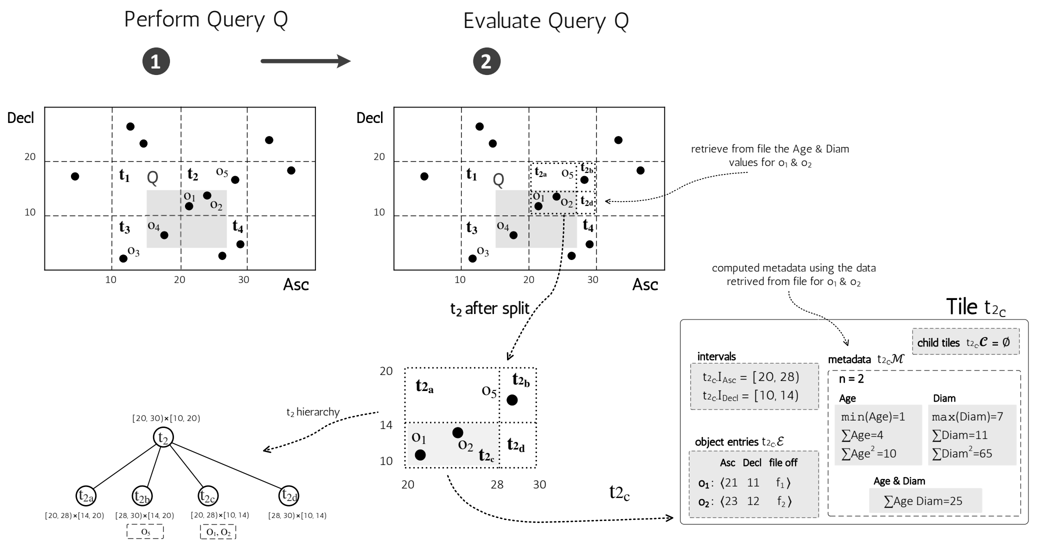


Figure 4: Query Processing over VALINOR Index

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 part. 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. 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. 5.2, Ex. 5). ■

4.3. *Select Part Evaluation*

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

First, we define the following simple function used in the Select part evaluation.

– getSelectOverlappedLeafTiles($\mathcal{I}$, S): This function returns the leaf tiles $\mathcal{T}_\mathsf{S}$ which overlap with the Select part 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 Select part S of the query.

In case that the tile t is fully-contained in the Select part, 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.

```
Procedure 1: evaluateSelectPart($\mathcal{I}$, S)
  Input: $\mathcal{I}$: index;    S : Select part of the query
  Output: $O_\mathsf{S}$: objects selected from Select part;    $\mathcal{T}_\mathsf{S}$: leaf tiles that overlapped with Select part

1 $\mathcal{T}_\mathsf{S} \leftarrow$ getSelectOverlappedLeafTiles ($\mathcal{I}$, S)
2 forall $t \in \mathcal{T}_S$ do
3     $t.\mathcal{E}_\mathsf{S} \leftarrow$ getSelectedObjectsFromTile ($t$, S)
4     insert $t.\mathcal{E}_\mathsf{S}$ into $O_\mathsf{S}$
5 return $O_\mathsf{S}$, $\mathcal{T}_\mathsf{S}$
```

```
Procedure 2: getTilesRequireFileAccess($\mathcal{T}_\mathsf{S}$, $Q$)
  Input: $\mathcal{T}_\mathsf{S}$: leaf tiles that overlap with the Select part of the query;    $Q$ ⟨S, F, D, N⟩: query
  Output: $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$: leaf tiles that require file access

1  forall $t \in \mathcal{T}_S$ do
2      if $t \in \mathcal{T}_{S_f}$ 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 part 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 part 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 parts
9              $accessRequired \leftarrow$ false
10         else if $N$ and $F$ can be evaluated using $t.\mathcal{M}$ then    //Filter and/or Analysis part 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 $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$
```

The evaluation of Select part is described in the evaluateSelectPart 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 Select part of the query (*line* 3). Finally, the evaluateSelectPart procedure returns the objects O_S selected from the Select part and the leaf tiles $\mathcal{T}_\mathsf{S}$ (*line* 5).

4.4. 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 part is defined, or a Filter is included, and its condition can not be computed using metadata. On the other hand, if tile is

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

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

partially-contained (*line* 5), in case that a Details part is defined in the query (*line* 6), we always have to read from the file the values of the objects included in the Details part. Also, we have to examine if the computations defined in the Analysis and Filter parts can be evaluated using the metadata that are already available in each tile (*line* 10).

4.5. Progressive Index Adaptation

During query evaluation, we employ an *progressive 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 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 5), .

In a baseline implementation introduced in [20], 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.

4.6. 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 Select part, 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 part (*line* 3).

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

```
Input: $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$: tiles for which file access is required;   $O_S$: objects included in Select part;
       $Q\,\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$: 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 part

1 forall $o_i$ included in tiles $\mathcal{T}_{\mathsf{S}_\mathcal{F}}$ with $o_i \in O_S$ do
2     access $\mathcal{F}$ at file offset $f_i$
3     $\mathcal{V}_{\mathsf{F}_{A i}}$, $\mathcal{V}_{\mathsf{D}_i}$, $\mathcal{V}_{\mathsf{N}_{A i}}$, $\leftarrow$ read the $o_i$ attributes values that are required for the F, D and N parts
4     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 $\mathcal{V}_{\mathsf{N}_A}$;
5 return $\mathcal{V}_{\mathsf{F}_A}$, $\mathcal{V}_\mathsf{D}$, $\mathcal{V}_{\mathsf{N}_A}$
```

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 $\times$ 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.).

4.7. Aggregate Metadata Management

The metadata is used to improve the performance of queries with an Analysis and/or Filter part, 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 parts 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 part 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.

4.8. Filter, Details & Analysis Parts Evaluation

In the general case, the *Filter part* 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 O_S objects for filtering out the query objects O_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 part*, 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[9]. For the objects O_Q we retrieve the values $\mathcal{V}_{\mathsf{D}}$ of the attributes included in the Details part (Alg. 2, *line* 5). Then, for each object of 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 part* 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 parts operate as traditional selection operations on the data (the Select part is evaluated over the two axis attributes, whereas the Filter part 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).

5. 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 parts of the query usually require access to the raw file to fetch the values for the extra attributes involved in these parts. 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 Select part 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., I/Os cost), nor identify $t.\mathcal{E}_{\mathsf{S}}$ (i.e., computation cost). Hence, fully-contained tiles reduce both computation and I/Os cost (for more details see Sect. 5.3).

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.

[9]Note that, for both Filter and Details parts 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.

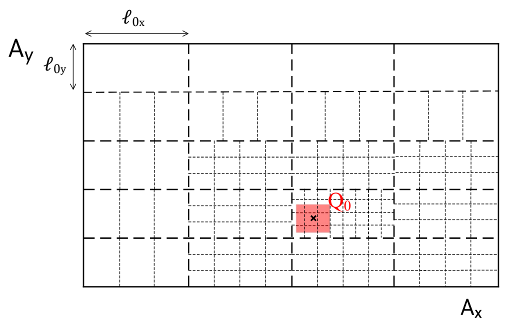


Figure 5: Query-Driven Index Initialization

5.1. Query-driven Index Initialization

This section presents an advanced approach for the initialization of the VALINOR index. The baseline initialization policy, as introduced in [20], groups objects into equal-size tiles but does 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 *Select part*, *lying in the tile* t_0 and continues the exploration by applying the set of visual operations presented in Section 2. 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 5, 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. 5.2) the method provides fast results *for the entire user session*.

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 $\ell_{0_x}/2 \times \ell_{0_y}/2$.

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:

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

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.

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 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) [50, 14, 78], 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.[10] 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.

[10]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).

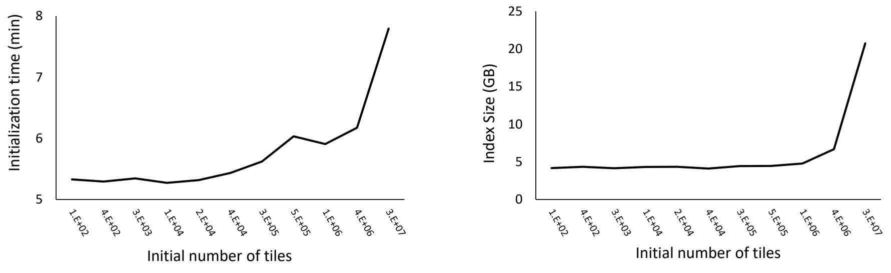


Figure 6: Initialization Time & Index Memory Size varying the Number of Initial Tiles

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 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 6 presents the initialization time and index size in relation to the number of initial tiles for a synthetic dataset SYNTH10 (see Sect. 7). 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 [50, 14, 78, 21]. However, this is beyond the scope of this work.

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 $mem(\mathcal{T}_0) = \sum_{t \in \mathcal{T}_0} mem(t) = |\mathcal{T}_0| \cdot b_t + b_o \cdot |O|$, whereas after splitting the index footprint becomes $mem(\mathcal{T}) = \sum_{t \in \mathcal{T}} mem(t) = |\mathcal{T}_0| \cdot b_t + |\mathcal{T}_S| \cdot b_t + b_o \cdot |O|$, 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., $|\mathcal{T}_S| \leq (mem_{MAX} - |\mathcal{T}_0| \cdot b_t - b_o \cdot |O|)/b_t$.

Note that, as $|O| \gg |\mathcal{T}|$, 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 6, we provide an eviction method for handling cases where the size of objects in memory do not fit in the allocated memory resources.

5.2. *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 4.5. As presented, in Section 4, tiles visited by the query can be split into smaller ones, i.e., *the index is progressively adapted to the user's interaction*. The index adaptation performs tile splitting, computes metadata and reorganizes objects into smaller groups during use 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 presented in [20] 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.

Query-driven Tile Splitting Overview. We consider a query which contains an Analysis part; 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_t^{S} that overlaps with the query, we retrieve the attribute values that are required for the Analysis part. Then, we have to compute the metadata for the area R_t^{S}, 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_t^{S} 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 2, after evaluating the query Q and adapting the index (Fig. 4), a subsequent query Q' is performed, as presented in Figure 7. 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 2, in order to evaluate Q', we have to examine the overlapping tiles and

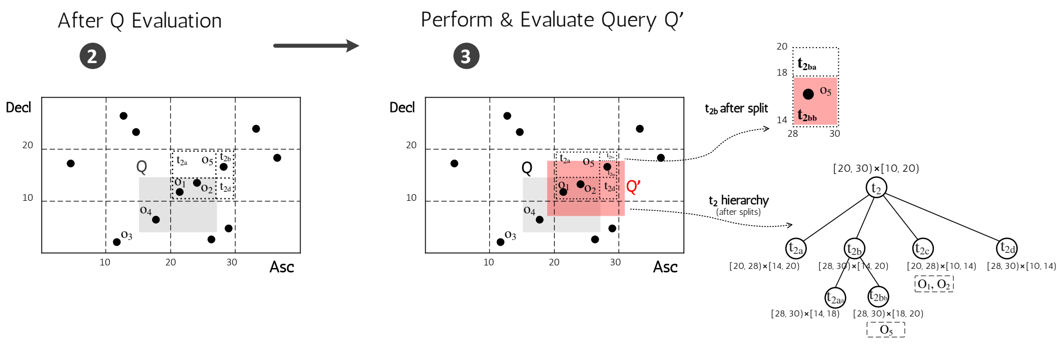


Figure 7: Index Adaptation and Query Processing

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 2, 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 file. ■

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_t^{S} 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_t^{S}. In Figure 7, at the left, the *query subtile* corresponds to t_{2_c}.

Here, for ease of presentation, given a query Q, the intervals of the Select part $\mathsf{S}.I_x$ 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 Select part 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 8 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. 8, 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: $(\mathbf{t_Q})$ $t_{Qx} = [t_{x1}, t_{x2}]$, $t_{Qy} = [t_{y1}, Q_{y2}]$; $(\mathbf{t_a})$ $t_{ax} = [t_{x1}, t_{x2}]$, $t_{ay} = [Q_{y2}, t_{y2}]$.

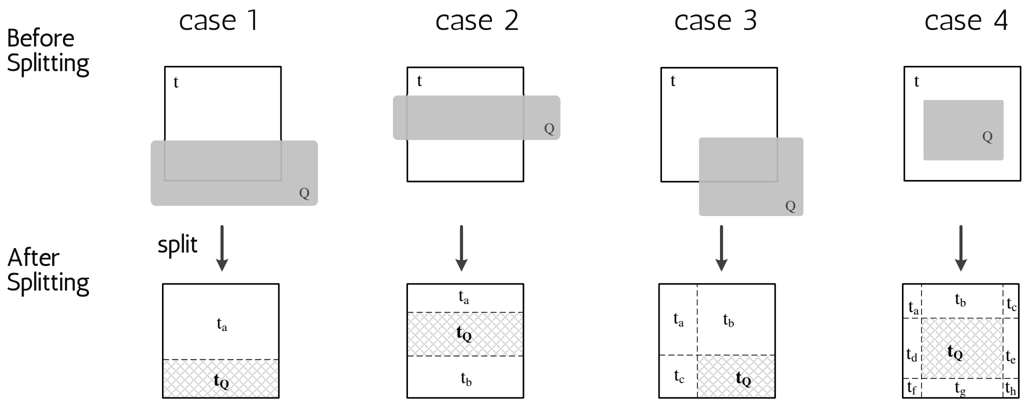


Figure 8: Tile Splitting Cases

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

– 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: ($\mathbf{t_Q}$): $t_{Qx} = [Q_{x1}, Q_{x2}]$, $t_{Qy} = [Q_{y1}, Q_{y2}]$; ($\mathbf{t_a}$) $t_{ax} = [t_{x1}, Q_{x1}]$, $t_{ay} = [Q_{y2}, t_{x2}]$; ($\mathbf{t_b}$) $t_{bx} = [Q_{x1}, Q_{x2}]$, $t_{by} = [Q_{y2}, t_{y2}]$; ($\mathbf{t_c}$) $t_{cx} = [Q_{x2}, t_{x2}]$, $t_{cy} = [Q_{y2}, t_{y2}]$; ($\mathbf{t_d}$) $t_{dx} = [t_{x1}, Q_{x1}]$, $t_{dy} = [Q_{y1}, Q_{y2}]$; ($\mathbf{t_e}$) $t_{ex} = [Q_{x2}, t_{x2}]$, $t_{ey} = [Q_{y1}, Q_{y2}]$; ($\mathbf{t_f}$) $t_{fx} = [t_{x1}, Q_{x1}]$, $t_{fy} = [t_{y1}, Q_{y1}]$; ($\mathbf{t_g}$) $t_{gx} = [Q_{x1}, Q_{x2}]$, $t_{gy} = [t_{y1}, Q_{y1}]$; ($\mathbf{t_h}$) $t_{hx} = [Q_{x2}, t_{x2}]$, $t_{hy} = [t_{y1}, Q_{y1}]$.

5.3. Splitting Model Analysis

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

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.*

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:

$$C_{part}^{Q}(t) = t.\mathcal{E} + c_{io} \cdot t.\mathcal{E}_{\mathsf{S}} \tag{1}$$

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 Select part S of the query. This is the cost of getSelectedObjectsFromTile function described in Section 4.3. 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:

$$C^{Q}_{full}(t) = 0 \tag{2}$$

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. 5.2). 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.5.2). Therefore, *the cost for splitting a tile t* is: $9 \cdot t.\mathcal{E}$.

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.[11] 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. 1 the *evaluation cost $\Phi^{Q,Q'}_{nosplit}$* of *Q' in case of no split*:

$$\Phi^{Q,Q'}_{nosplit} = C^{Q'}_{part}(t) = t.\mathcal{E} + c_{io} \cdot t.\mathcal{E}_{\mathsf{S}'} \tag{3}$$

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. 1, 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:

$$\Phi^{Q,Q'}_{split} = (t.\mathcal{E} - t.\mathcal{E}_{\mathsf{S}}) + c_{io} \cdot (t.\mathcal{E}_{\mathsf{S}'} - t.\mathcal{E}_{\mathsf{S}}) + (9 \cdot t.\mathcal{E}) + 9 \tag{4}$$

[11]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.

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. 3 & 4, the *expected splitting gain* $\Delta\Phi_{Q'}$ for the query Q' is defined as:

$$\Delta\Phi_{Q,Q'} = \Phi^{Q,Q'}_{nosplit} - \Phi^{Q,Q'}_{split} = c_{io} \cdot t.\mathcal{E}_{\mathsf{S}} \tag{5}$$

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.

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. 5 we have:[12]

$$\textit{Splitting Criterion}: \text{ if } (c_{io} \cdot t.\mathcal{E}_{\mathsf{S}}) > \epsilon\,, \quad \text{perform a split} \tag{6}$$

We can observe in Eq. 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.

6. 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.

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 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.

[12]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.

Each time a tile is selected to be evicted, all of its objects *currently residing in memory* are written to the disk[13] 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 $N = |O|$ the number of objects in the dataset. Further, we assume that the main memory can fit M objects[14], with $N > M$.[15]

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.

6.1. 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. 5.3). 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.

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 [50, 14, 78, 21]. 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.

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

[14]Section 5.1 presents the memory requirements of a tile and an object.

[15]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.

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

$$\mathbb{E}_t = t_{evSc} \cdot C_{io}(t) \tag{7}$$

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:

$$\text{minimize} \sum_{\forall t_i \in \mathcal{T}_e} \mathbb{E}_{t_i} \quad \text{subject to} \sum_{\forall t_i \in \mathcal{T}_e} |t_i.\mathcal{E}| \geq V \tag{8}$$

Selecting Tiles to be Evicted. Considering the objective of the eviction policy (Eq. 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.

6.2. 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.

Table 3: Datasets Basic Characteristics

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

7. Experimental Analysis

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

7.1. Experimental Setup

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[16], 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 3 presents the basic characteristics of the datasets. In our experiments for the TAXI dataset, the 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 $\times$ 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.

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.

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

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[17], 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 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. [87, 88, 52, 78, 14, 84, 28, 31]. 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 [14] for 2D exploration, the users perform almost equal number of pan and zoom operations. Further studies [72] 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.

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 the basic initialization mode with the basic quad-tree like adaptation mode as introduced in [20], while in the third (VALINOR), we use the query-based initialization mode (Sect. 5.1) with the query-driven adaptation mode (Sect. 5.2). 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.

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

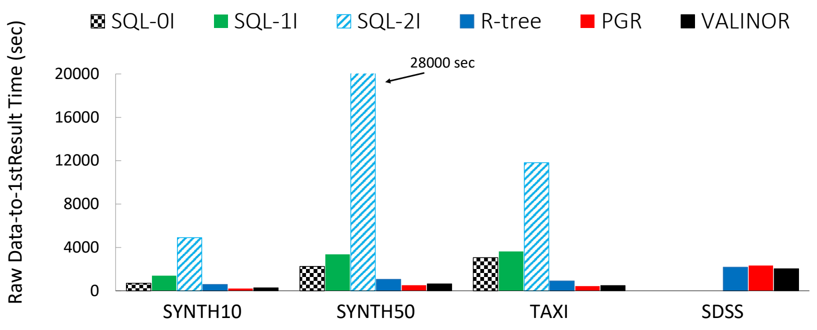


Figure 9: Time for Answering the 1st Query over the Raw File.
Time includes: File Parsing, Index Construction & Q_0 Evaluation

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 Sect. 8); however, due to low performance [9], we do not consider it as a competitor in our evaluation[18]. (2) PostgresRaw (PGR)[19], build on top of Postgres 9.0.0 [9], which is a generic platform for in-situ querying over raw data (Sect. 8). (3) A main memory Java implementation of the R*-tree[20] [15]. 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.

Metrics. We compare our method with the existing solutions, as well as with our previous baseline approach of [20]. 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.

Implementation. We have implemented RawVis[21] 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, PGR 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. 7.2).

[18]We refer the reader to [9], 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.

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

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

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

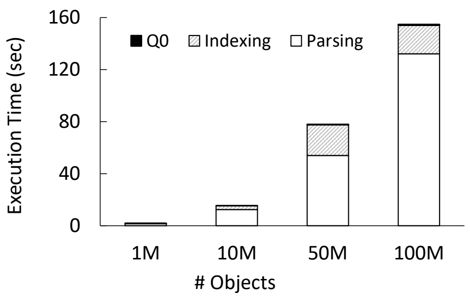


Figure 10: Initialization Phase: File Parsing, Index Construction & Q_0 Evaluation

7.2. Results

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 PGR. For the VALINOR and R-tree cases the in-memory indexes must be built. For the R-tree construction, bulk-loading was used.

Figure 9 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 PGR 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 PGR (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$\times$ 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 analysis part of the queries. PGR, 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, PGR 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

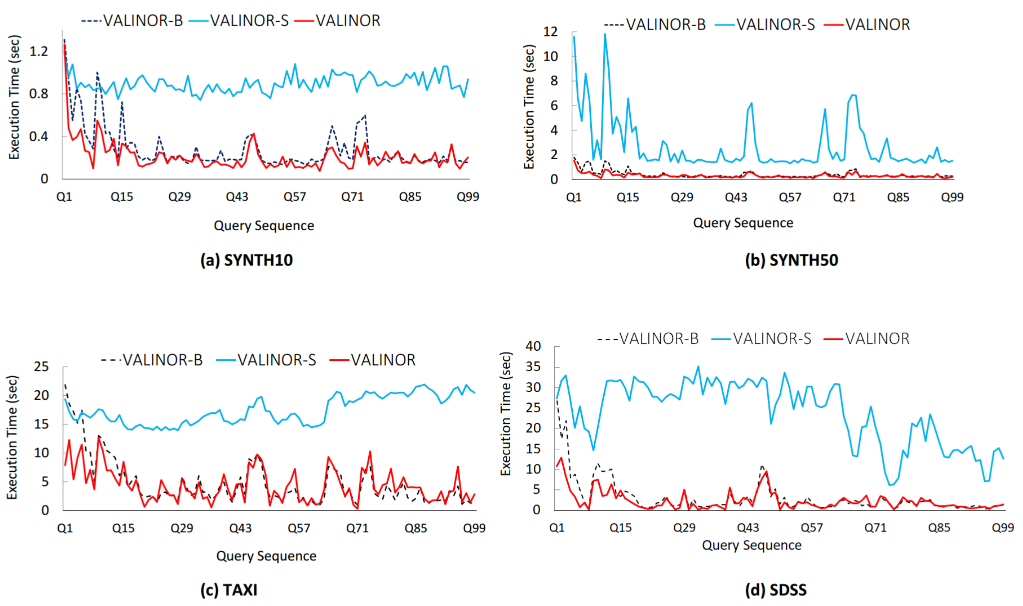


Figure 11: Initialization & Adaptation Methods: Execution Time Comparison of the Three VALINOR Configurations

use the SYNTH10 data varying the number of objects from 1M to 100M objects. The results are presented in the Figure 10. 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.

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 11 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-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

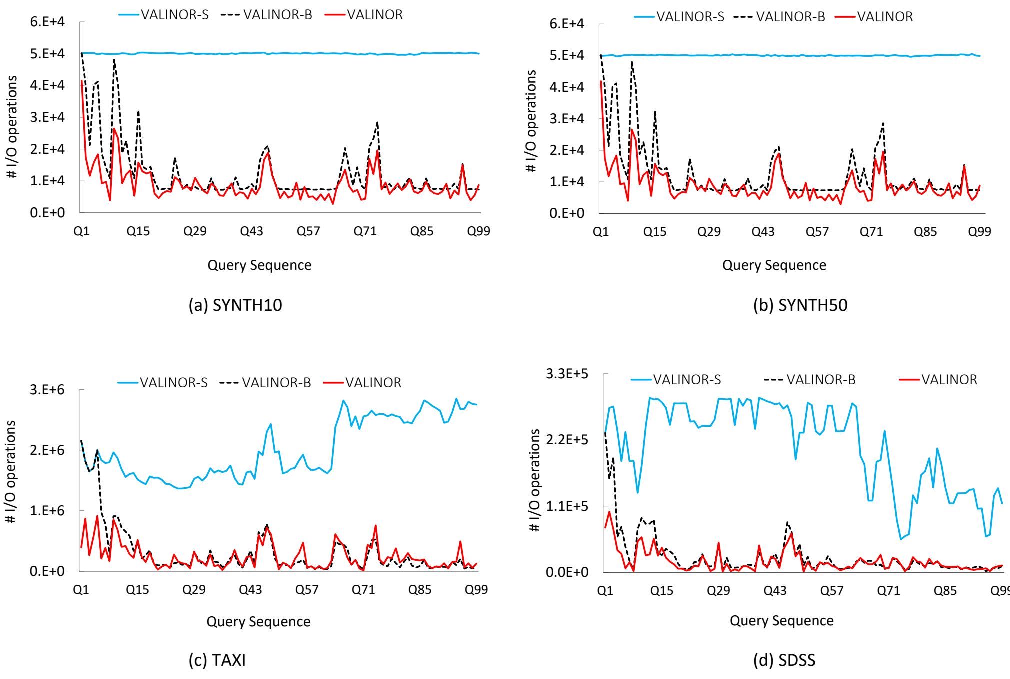


Figure 12: Initialization & Adaptation Methods: Number of I/O Operations Comparison of the Three VALINOR Configurations

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 11, 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 12, where as it can be seen, the plots follow closely the corresponding execution time plots in Figure 11. Regarding the two synthetic datasets (SYNTH10/50), their I/O plots almost completely match (Fig. 12a, 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.

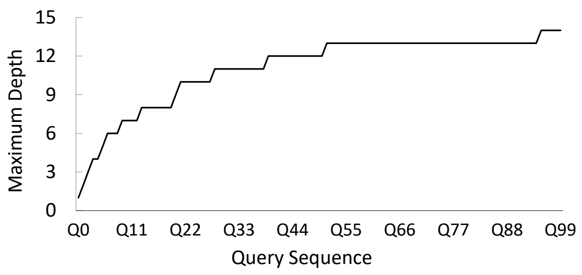


Figure 13: Maximum Hierarchy Depth of VALINOR per query (TAXI)

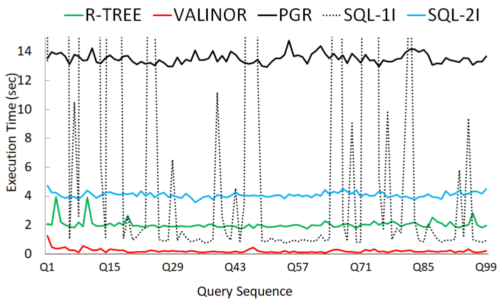
Regarding the index adjustment to the query selection predicate, the incremental index adaptation performs a larger number of tile splittings in areas that are frequently visited by the user. As a result, an unbalanced index is constructed, with deeper tile hierarchies in those areas. On the other hand, the threshold used by our splitting method (Sect. 5.3) limits the number of times a tile is split. Figure 13 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 $Q_1 \sim Q_{99}$ 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 $Q_1 \sim Q_{99}$. 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 15 shows that although Q_0 size affects the initial tile structure, VALINOR attempts to adjust the number 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.

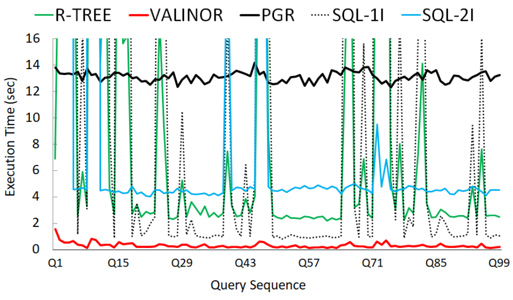
VALINOR vs. Competitors during Exploration Scenarios. In this experiment, we compare the behavior of VALINOR against the existing solutions. Figure 14 shows the execution time for queries $Q_1 \sim Q_{99}$, without the first query that includes the initialization stages for every system (e.g., loading 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.

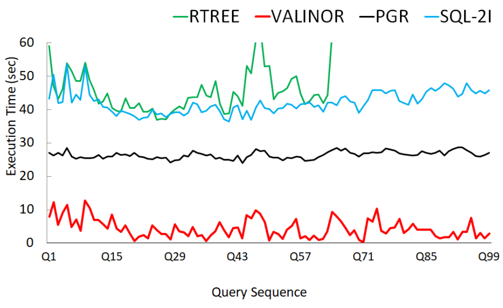
Compared to the other methods, VALINOR exhibits significantly lower execution time in almost all cases. Particularly in TAXI dataset, where VALINOR times range between 0.3 to 12 sec, VALINOR is more than 2× faster in all queries and more than 10× faster in 35% of queries than the best competitor, and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



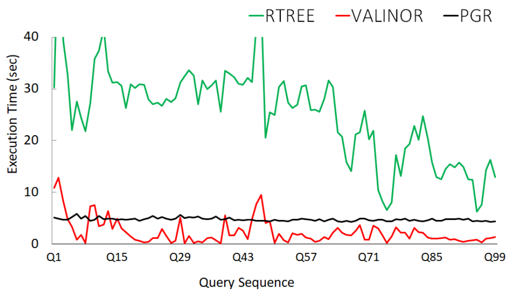
(a) SYNTH10



(b) SYNTH50



(c) TAXI



(d) SDSS

Figure 14: Execution Time: VALINOR vs. Competitors

Regarding PGR, we observe that it requires approximately the same time for every query. The positional map used in PGR, 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, PGR 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, PGR 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 part 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 select part of the query.

Besides the positional map used in PGR, 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, PGR 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 PGR 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 PGR for most of the queries in SDSS, needing approximately 51% less total time to execute queries $Q_1 \sim Q_{99}$.

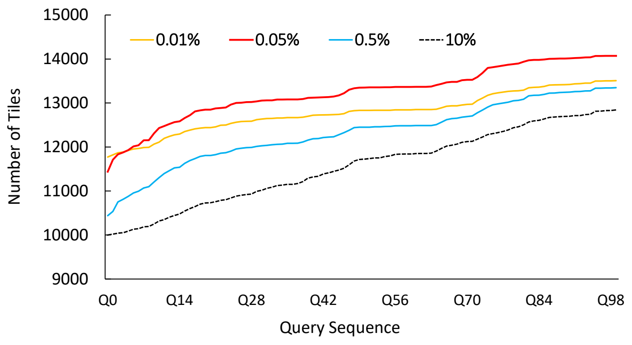


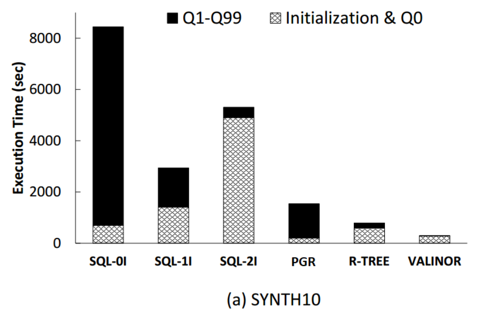
Figure 15: Number of Tiles varying Q_0 Selectivity

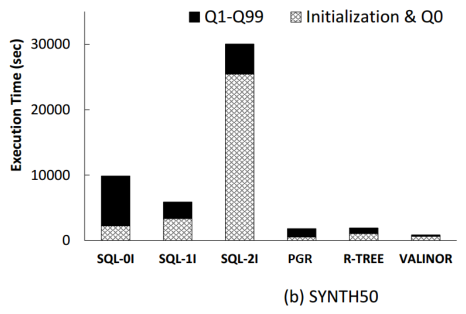
The accumulative time needed to execute the query sequence of the exploration scenario for every dataset is shown in Figure 16. The accumulative time captures the overall performance of the user scenario. This time includes Q_0 which is depicted separately from all subsequent queries. As it can be seen, 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 PGR, which is the best competitive method for this dataset, requires approximately 51 min. Even though PGR 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.

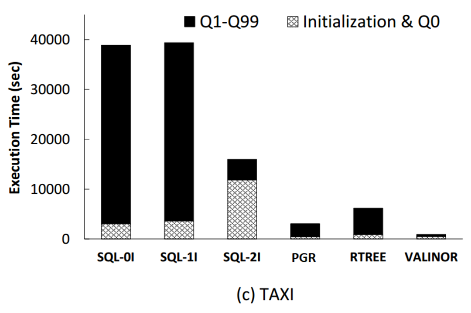
Discussion. We observe that VALINOR achieves for most queries (except Q1) of SYNTH10 and SYNTH50 response times between 0.07 and 0.55 and between 0.1 and 0.8, respectively. Note that, in SYNTH10 *only one query reports time more than 0.5 sec*, and in the SYNTH50 dataset 11 queries. Regarding the SDSS 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 (PGR), 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 (PGR), reports times between 23 and 28, with 26 as average value. Further, in PGR about 85% of the queries require more than 25 sec. Hence, in 85% of the queries the PGR 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.

Evaluating Filter Operations. For assessing the behaviour of VALINOR with regard to varying filtering on non-axis attributes, we compare VALINOR against PGR while varying the filter part. For this, we generate 4 queries for SYNTH10, keeping the select part (i.e., window query) fixed, while alternating their filter condition. Specifically, the filter part of each query includes







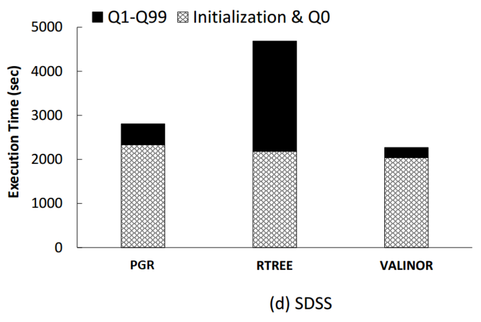


Figure 16: Overall Execution Time for the Entire Exploration Scenario

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 $Q_0 \sim Q_3$ are repeated 3 times and the results are shown in Figure 17. 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 PGR in this experiment. For every such query, VALINOR first evaluates the select part, and may read the raw file to retrieve the non-axis attribute included in the filter part 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 select part. 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 PGR, 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, PGR also exhibits a significantly slower execution time for the first iteration of $Q_1 \sim Q_3$ 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 PGR.

Combining Pan & Zoom Operations. In this experiment, we compare VALINOR's performance with that of PGR and R-tree, for the second exploration scenario which includes pan, zoom-in and zoom-out operations on the TAXI dataset. Figure 18 presents the results, where

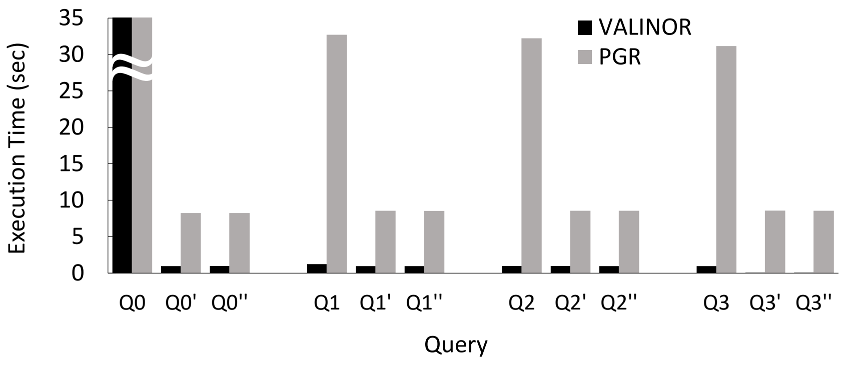


Figure 17: Execution Time for Filter Operations (SYNTH10)

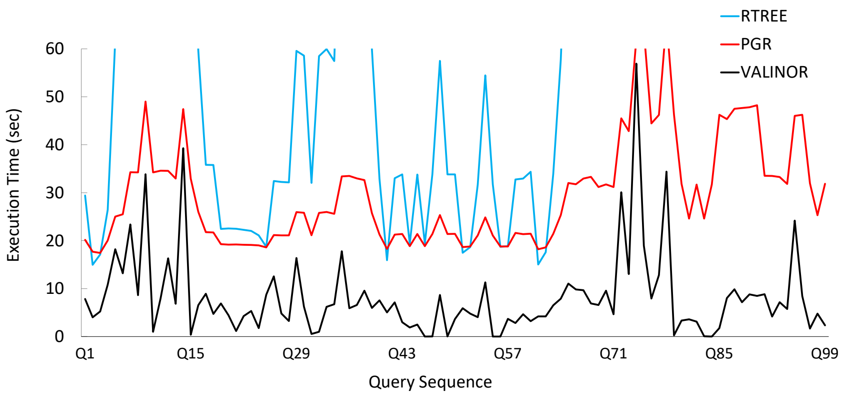
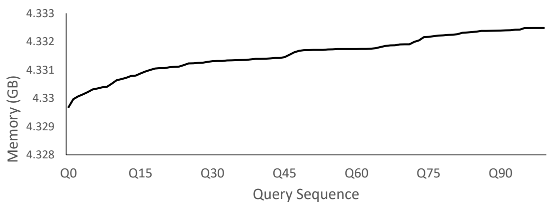


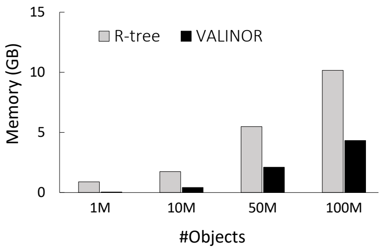
Figure 18: Exploration using Pan and Zoom In/Out Operations (TAXI)

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 PGR and R-tree methods, where all contained objects have to be examined and their non-axis attributes included in the analysis part of the query, either fetched from disk for R-tree, or from disk or cache for PGR. 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.

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



(a) VALINOR During an Exploration Scenario



(b) VALINOR vs. R-tree

Figure 19: Memory Consumption (SYNTH10 or SYNTH50 – is the same in both datasets–)

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 19(a) 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 19(b) presents VALINOR's memory footprint compared to R-tree. We did not consider PGR and MySQL settings since they exhibit different memory requirements due to their tight-coupling with the RDBMS. Nevertheless, PGR required a significant amount of memory for its positional map and 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.

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 20 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 processing, we present separately in Figure 21(a) the cumulative time needed to answer $Q_1 \sim Q_{99}$. 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

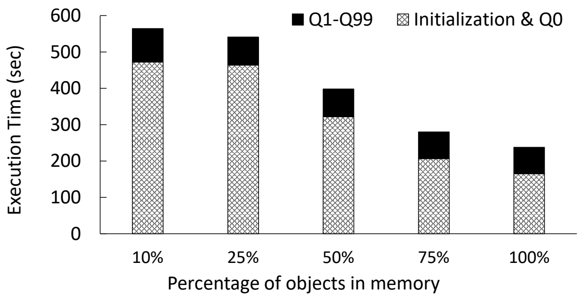
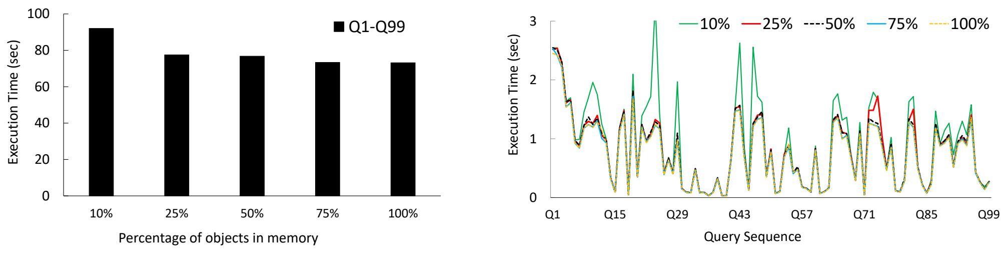


Figure 20: Overall Execution Time varying the Memory Size (SYNTH10)



(a) Overall Execution Time

(b) Execution Time per Query

Figure 21: Execution Time varying the Memory Size (SYNTH10)

more pronounced when restricting the available memory to 10% of the dataset, as can also be seen in Figure 21(b), which presents the execution time for queries $Q_1 \sim Q_{99}$.

8. Related Work

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. In this respect there are recent works that aim to compute and store statistics for the data on-the-fly minimizing the access to the raw data. Also, several spatial indexes 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.

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 [44]. In-situ query processing aims at avoiding data loading in a DBMS by accessing and operating directly over raw data files. NoDB [9] is a philosophy for constructing a no-dbms querying architectures, 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. As opposed to VALINOR, the positional map in PostgresRAW, can only be exploited to reduce parsing and tokenization costs during query evaluation and can not be used to reduce the number of objects examined in two-dimensional range queries. Also, VALINOR is better optimized for aggregate queries, since it can reduce raw file accesses by reusing already calculated statistics on a tile level.

DiNoDB [79] is a distributed version of PostgresRAW. In the same direction, PGR [55] extends the positional maps in order to both index and query files in formats other than CSV. In the same context, Proteus [54] supports various data models and formats. Recently, Slalom [67, 68] 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 [49] and SDS/Q [22] rely on DBMS technologies to perform analysis over scientific array-based file formats. Further, SCANRAW [26] considers parallel techniques to speed up CPU intensive processing tasks associated with raw data accesses.

Recently, several well-known DBMS support SQL querying over csv files. Particularly, MySQL provides the CSV Storage Engine [1], Oracle offers the External Tables [3], and Postgres has the Foreign Data [4]. However, these tools do not focus on user interaction, parsing the entire file for each posed query, and resulting in significantly low query performance [9] for interactive scenarios.

All the aforementioned works study the generic in-situ querying problem without focusing on the specific needs for raw data visualization and exploration. Instead, our work is the first effort trying to address these aspects, considering the in-situ processing of a specific query class, that enables user operations in 2D visual exploration scenarios; e.g., pan, zoom, details. The goal of our solution is to optimize these operations, such that visual interaction with raw data is performed efficiently on very large input files using commodity hardware.

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 [70, 17, 11, 37]. In this context, the first efforts focused on developing visual querying languages for DBs such as [90, 12, 24, 25, 62]. Although, they share some similar concepts, most of them 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 [30, 41, 42, 83].

In the context of visual exploration, several indexes have been introduced. VisTrees [32] and HETree [21] 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), and they do not consider disk storage; i.e., data is stored in-memory.

Hashedcubes [29], Nanocubes [59], SmartCube [60] and Gaussian Cubes [80] and are main-memory data structures supporting a wide range of interactive visualizations, such as heatmaps, time series, and histograms. They are based on main-memory variations of a data cube in order to reduce the time needed to generate the visualization. In comparison with our approach, Hashedcubes and Nanocubes require that all data resides in memory, and thus it does not address the need of reducing the overall time-to-visualization (both loading and query processing time) over raw data files and it does not feature any adaptive technique based on the user interaction. Top-Kubes [63] proposes an extension of Nanocubes for the interactive computation of top-k queries in large datasets.

Further, graphVizdb [19, 18] 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 our 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.

In another context, tile-based structures are used in visual exploration scenarios. Semantic Windows [52] 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 [14] 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. Our 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). However, these issues are beyond the scope of this work.

Recently, many systems adopt the *progressive paradigm* attempting to reduce the response time. [33, 13, 73, 86, 8, 35]. 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. On the other hand, in our approach, the exact answers are presented to the users as soon as these are computed.

Exploratory Data Analysis. 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 [71] or NumPy and SciPy [5]. These tools cannot handle our scenario, though, since they either assume that the data fits in memory, or are integrated with traditional database systems which require a preprocessing phase.

In [82], Data Canopy is introduced, which attempts to reduce the number of data accessed while calculating statistics, by synthesizing statistics from basic aggregates calculated over chunks of the data columns and are cached for reuse by future queries. Although, Data canopy, like VALINOR allows the reuse of already cached basic aggregates for the efficient calculation of more general statistics, it does not deal with the problem of fast exploration over large raw datasets. Also, in Data Canopy, the chunks are defined over consecutive data items from a column or a set of columns, as opposed to VALINOR where the data objects are organized into tiles based on their values for the two axis attributes. In that way, the chunking used in Data Canopy can be used to compute statistics over query ranges defined between two positions in a column set and can not be exploited for the evaluation of two-dimensional window queries.

Traditional and Adaptive Indexes. A vast collection of index structures has been introduced in traditional databases, as well as Big Data systems. Traditional spatial indexes, such as the R-tree, kd-tree, quadtree [36], are designed to improve the evaluation of range or nearest-neighbor queries on multidimensional data, and are widely available in both disk-based and main memory implementations. In R-trees [61], 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 [16] 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 [27], 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 [61], 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 (see also the results in Sect. 7). On the contrary, our approach proposes a main-memory lightweight index, which aims at accelerating the raw data-to-visualization time and offering a simple set of 2D visual operations to the user, rather than covering aspects of spatial data management.

[88] 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 [87]. To this end, a method that selects which part of the data to be cashed 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 [89]. Similarly to our work, [89] considers raw data files, as well as the incremental indexing paradigm. However, our work considers different settings, problems and goals. For example, our focus is to provide 2D in-situ visual exploration over raw file, without considering a caching mechanism or a distributed setting. Further, our index structure is used to store file positions and compute aggregate functions; instead, in the aforementioned work, the index stores data objects (not file positions). Additionally, the basic goal of our index is to minimize I/O operations, as well as statistics computation. Notwithstanding the differences, the introduction of concepts and ideas from [88] in VALINOR is a topic of future work.

Similarly to VALINOR, the basic idea of approaches like database cracking and adaptive indexing [40, 38, 45, 47, 46, 69, 43, 74, 65, 43, 10], is to incrementally build and adapt indexes 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. In addition, the existing cracking and adaptive indexing methods have been developed in the context of column-stores [40, 38, 45, 47, 46, 69, 10], or MapReduce systems [74]. On the other hand, VALINOR has been developed to handle raw data stored in text files with commodity hardware. Due to the common adaptive nature of the techniques, the introduction of concepts and ideas of data cracking into VALINOR is a topic of future research.

Finally, our structure employs a value-existence indexing technique that is similar to the concept of the Zonemaps [64, 2], which has been widely used in traditional databases. Each cell in the grid groups the positions of the records in the raw file based on their distance in the 2D plane, keeping the bounds and other statistics (e.g., count) in the form of cell properties. Range queries are quickly evaluated on the cell properties (zones) and only the cells that overlap the query ranges are further processed, thus eliminating unnecessary memory accesses.

Grid Files –and a critical discussion– . The basic structure of the presented index has several similarities with the grid file [66]. Both partition the space, organizing the data objects in tiles/cells. Basic differences between them are related to the tiles' merging and splitting phases, methods, and criteria. Firstly, in grid both the merging/splitting phases are performed during the grid construction. On the other hand, in VALINOR the merging/splitting operations are performed after construction, during the runtime. Further, in VALINOR the merging/splitting is performed incrementally and adaptively, based on user interactions. Secondly, the criteria which are adopted to determine the merging/splitting, are different. In a grid file, the merging/splitting is based on criteria like better storage utilization, min/max number of objects per tile, number of I/O accesses, budget size, etc. In contrast, in VALINOR the merging/splitting is based on the user's interaction (queries). Furthermore, in VALINOR the initial structure characteristics (e.g., tile size) are estimated by the first user's interaction. Also, VALINOR computes and exploits specific metadata focusing on improving visual-based operations and analytics.

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 [66]. 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 [76, 85, 77]. Further, some studies suggest that grid indexes have better performance even over the R-tree versions that use efficient bulk loading methods [85, 77]; i.e., STR [58] and Hilbert R-Tree [53].

Regarding the query performance, recent studies [75, 51, 76], show that the grid index, have noticeable better performance in range and kNN queries, as well updates operations, compared to R-tree variances and quadtree, *when the indexing is performed in 2 dimensions and the indexes are stored in main memory*. Additionally, [76] concludes that grid index is surprisingly robust to varying parameters of the query workloads.

9. Conclusions and Future Directions

In this paper, we have presented the RawVis framework that is built on the top of the VALINOR index, a light weighted main memory structure, which enables interactive 2D visual exploration scenarios of very large raw data files using commodity hardware. VALINOR is constructed from a raw data file given the first user query and adapted based on the user interaction.

We have formulated a set of basic visual operations and mapped them to query operators evaluated on the VALINOR index. Further, we have designed an advanced initialization method in the context of visual exploration, which reduces the time of index construction, and in the same time, improves the query evaluation at the initial stages of the exploration. We have proposed a query-based adaptation model that restructures the index based on the user interaction, resulting in efficient computation of the analysis operations. Also, we have developed an eviction mechanism that stores parts of the index in disk in case of limited memory resources. Finally, we have conducted a thorough experimental evaluation with real and synthetic datasets and compared with several competitors (e.g., MySQL, PostgresRaw, R-tree). The results demonstrates that our methods outperform the competitors in query execution time, number of I/O operations, and memory consumption.

In what follows we mention two basic directions - in the form of research questions - which are not covered in this work but worth further investigation in our future studies.

How can we smoothly scale up dimensionality? For the moment, our approach is built upon the assumption that the user interacts with two attributes (the axis attributes) of the data as the basis of the exploration. This is reasonable to a large extent and well known for long [81, 34], as the 2.05-dimensional nature of the human eye [81] and the 2-dimensional nature of the media (being paper or screen) practically limits the intuition offered by 3D models, let alone of higher dimensionality. Fundamental visual representations like tabular, scatter-plot or bar-chart-with-data-series representations are practically at the very end of the visualization spectrum that starts with simple KPI reporting via numbers or speedometers, passes via the traditional point/bar/pie charts and ends at the aforementioned visual representations. However, using more attributes for visual exploration can have a potential in the future. To support this exploration scenario, VALINOR will have to be enriched with a cache that will maintain previously accessed attributes, besides the ones that were used to build it.

What if the data sets are not static, flat files? A clear limitation of this paper is that the data need to be stored in flat files on disk. In case of data with a complex –e.g., nested – structure, like for example JSON or XML data would need to be transformed first, assuming that this is feasible. Thus, there is an open research path, to accommodate non-flat raw data into VALINOR, without the need of preprocessing them first. Similarly, in this paper, we assume no updates to the data. The incremental maintenance of the index, in the presence of updates, is another open problem. Assuming that the users update the file without touching VALINOR, a potential path to follow would be the fast identification of insertions and deletions (updates in this setting can be modeled as a pair of 'delete old and insert new') [56], and the subsequent adaptation of the index accordingly.

What are the deep foundations of an algebra of visual operations? In this paper, we proposed a set of visual operations, very common to 2D visual exploration scenarios. However, we do not consider more complex or combinations of those operations which can support the application of more complex visual analytics. Assuming we want to equip the end users with the potential of interactively exploring the data, what kinds of operations constitute the fundamental core of operations, upon which more complex operations can be built? Do we need a closed algebra that can guarantee a scoped horizon of actions, or do we need extensible frameworks, where operations can be added along with the necessary infrastructure for their optimization?

Acknowledgments. The authors want to thank the reviewers and editors of this paper for their constructive comments that have significantly improved both the clarity and thoroughness of the paper. This research is co-financed by Greece and the European Union (European Social Fund- ESF) through the Operational Programme ”Human Resources Development, Education and Lifelong Learning 2014-2020” in the context of the project ”PLOIGIA: Navigation and Visual Analytics on Raw Datasets” (MIS 5005089).

References

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

[2] Oracle: Database Data Warehousing Guide - Using Zone Maps. `https://docs.oracle.com/database/121/DWHSG/zone_maps.htm`.

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

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

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

[6] 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`.

[7] Wolfram : Descriptive Statistics. `https://reference.wolfram.com/language/guide/DescriptiveStatistics.html`.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] N. Bikakis, S. Maroulis, G. Papastefanatos, and P. Vassiliadis. RawVis: Visual Exploration over Raw Data. In *Advances in Databases and Information Systems (ADBIS)*, 2018.

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

[22] 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.

[23] 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.

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

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

[34] S. Few. *Show Me the Numbers: Designing Tables and Graphs to Enlighten*. Analytics Press, 2012.

[35] 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.

[36] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2), 1998.

[37] P. Godfrey, J. Gryz, and P. Lasek. Interactive Visualization of Large Data Sets. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 28(8), 2016.

[38] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *Intl. Conf. on Extending Database Technology (EDBT)*, 2010.

[39] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pi-

rahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. *Data Min. Knowl. Discov.*, 1(1), 1997.

[40] 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.

[41] P. Hanrahan. VizQL: A Language for Query, Analysis and Visualization. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2006.

[42] J. Heer and M. Bostock. Declarative Language Design for Interactive Visualization. *IEEE Trans. Vis. Comput. Graph. (TVCG)*, 16(6), 2010.

[43] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt. Progressive Indexes: Indexing for Interactive Data Analysis. *PVLDB*, 12(13), 2019.

[44] 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.

[45] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *Conf. on Innovative Data Systems Research (CIDR)*, 2007.

[46] 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.

[47] 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.

[48] S. Idreos, O. Papaemmanouil, and S. Chaudhuri. Overview of Data Exploration Techniques. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2015.

[49] 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.

[50] 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.

[51] 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.

[52] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive Data Exploration Using Semantic Windows. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2014.

[53] I. Kamel and C. Faloutsos. On Packing R-trees. In *Intl. Conf. on Information and Knowledge Management*, 1993.

[54] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *VLDB Endowment*, 9(12), 2016.

[55] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on Raw Data. *VLDB Endowment*, 7(12), 2014.

[56] W. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *VLDB*, 1996.

[57] H. Lenz and B. Thalheim. OLAP Databases and Aggregation Functions. In *SSDBM*, 2001.

[58] 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.

[59] 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.

[60] 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. Vis. Comput. Graph. (TVCG)*, 26(1), 2020.

[61] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications*. Springer, 2006.

[62] A. Massari, S. Pavani, L. Saladini, and P. K. Chrysanthis. Qbi: Query by icons. In *ACM SIGMOD Record*, volume 24, 1995.

[63] 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.

[64] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Intl. Conf. on Very Large Databases (VLDB)*, 1998.

[65] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2020.

[66] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1), 1984.

[67] 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.

[68] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive partitioning and indexing for in situ query processing. *The VLDB Journal*, 2019.

[69] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2015.

[70] 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.

[71] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2018.

[72] P. Rahman, L. Jiang, and A. Nandi. Evaluating Interactive Data Systems. *VLDB J.*, 29(1), 2020.

[73] 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.

[74] 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.

[75] D. Sidlauskas and C. S. Jensen. Spatial Joins in Main Memory: Implementation Matters! *VLDB Endowment*, 8(1), 2014.

[76] 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.

[77] 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.

[78] F. Tauheed, T. Heinis, F. Schürmann, H. Markram, and A. Ailamaki. SCOUT: Prefetching for Latent Feature Following Queries. *VLDB Endowment*, 5(11), 2012.

[79] 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.

[80] 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. Vis. Comput. Graph. (TVCG)*, 23(1), 2017.

[81] C. Ware. *Visual Thinking: for Design*. Morgan Kaufmann, 2008.

[82] 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.

[83] E. Wu, L. Battle, and S. R. Madden. The Case for Data Visualization Management Systems. *VLDB Endowment*, 7(10), 2014.

[84] S. Yesilmurat and V. Isler. Retrospective adaptive prefetching for interactive Web GIS applications. *GeoInformatica*, 16(3), 2012.

[85] 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.

[86] 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.

[87] W. Zhao, F. Rusu, B. Dong, and K. Wu. Similarity Join over Array Data. In *ACM Intl. Conf. on Management of Data (SIGMOD)*, 2016.

[88] 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.

[89] 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.

[90] M. M. Zloof. Query-by-example: A data base language. *IBM systems Journal*, 16(4), 1977.