Noname manuscript No.
(will be inserted by the editor)

Resource-Aware Adaptive Indexing for In-situ Visual Exploration and Analytics

Stavros Maroulis · Nikos Bikakis · George Papastefanatos · Panos Vassiliadis · Yannis Vassiliou

Received: date / Accepted: date

Abstract In in-situ data management scenarios, large data files which do not fit in main memory, must be efficiently handled using commodity hardware, without the overhead of a preprocessing phase or the loading of data into a database. In this work, we study the challenges posed by the visual analysis tasks in in-situ scenarios in the presence of memory constraints. We present an *indexing scheme* and *adaptive* query evaluation techniques, which enable efficient categorical based group-by and filter operations, combined with 2D *visual interactions*, such as exploration of data points on maps or scatter plots. The indexing scheme combines a *tile-based structure*, which offers efficient visual exploration over the 2D plane, with a *tree-based structure* that organizes a tile's objects based on its categorical values. The index is constructed on-the-fly, resides in main memory and is built progressively as the user explores parts of the raw file, whereas its structure and level of granularity are adjusted to the user's exploration areas and type of analysis. To handle the cases where limited resources are available, we introduce a *resource-aware index initialization mechanism* and we formulate it as an NP- hard optimization problem; we propose two efficient approximation algorithms to solve it. We conduct extensive experiments using real and synthetic datasets, and demonstrate that our approach reports interactive query response times (less than 0.04sec); and in most cases is more than $100\times$ faster and performs up to 2 orders of magnitude less I/O operations compared to existing solutions. The proposed methods are implemented as part of an open-source system for in-situ visual exploration and analytics.

Keywords Big Data Visualization · Visual Analytics · Raw Data · Progressive Indexing · User-driven Processing

1 Introduction

A common task in data exploration scenarios involves *in-situ* visual data analysis, in which data scientists wish to *visually interact and analyze* large (and *dynamic*) raw data files (e.g., csv). These users usually have limited skills in data management and *limited resources* or *commodity hardware* for use (e.g., scientist's laptop), in contrast to, e.g., a distributed environment. In such scenarios, users need to perform the analysis directly over the raw files, avoiding the tedious tasks of loading and indexing the data in a data management system. Still, they expect a very small *data-to-analysis time* and they wish to interact via a rich set of *visual exploration and analytic operations*. To this end, efficient in-situ processing of raw files is a major challenge for a large number of real-word tasks over diverse domains, such as astronomy, physics, climatology, business intelligence, bioinformatics, finance, telco etc. We provide the following example to motivate our work and challenges.

Telecommunication Example. The data scientists working in telco companies analyze network data in order to get insights regarding the network quality. Such data are commonly stored in large comma-separated data files and contain signal and latency measurements crowdsourced from IoT mobile devices, e.g., connected cars, mobile phones.[1] Using this data, scientists visually explore, analyze and produce benchmarks regarding the network quality.

Figure 1(a) presents a sample of a raw file containing five entries/objects (o_1 - o_5). Each entry *represents a signal measurement* and contains information regarding the: *geographic location* (Lat, Long), *signal strength* (Signal) and *network bandwidth* (Width), as well as network and device characteristics which take *categorical values* such as: device *brand*, *network provider*, and *network technology* (Net).

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

⋆Under Submission

S. Maroulis
Nat. Tech. Univ. of Athens & ATHENA Research Center, Greece

N. Bikakis and G. Papastefanatos
ATHENA Research Center, Greece

P. Vasssiliadis
Univ. of Ioannina, Greece

Y. Vassiliou
Nat. Tech. Univ. of Athens, Greece

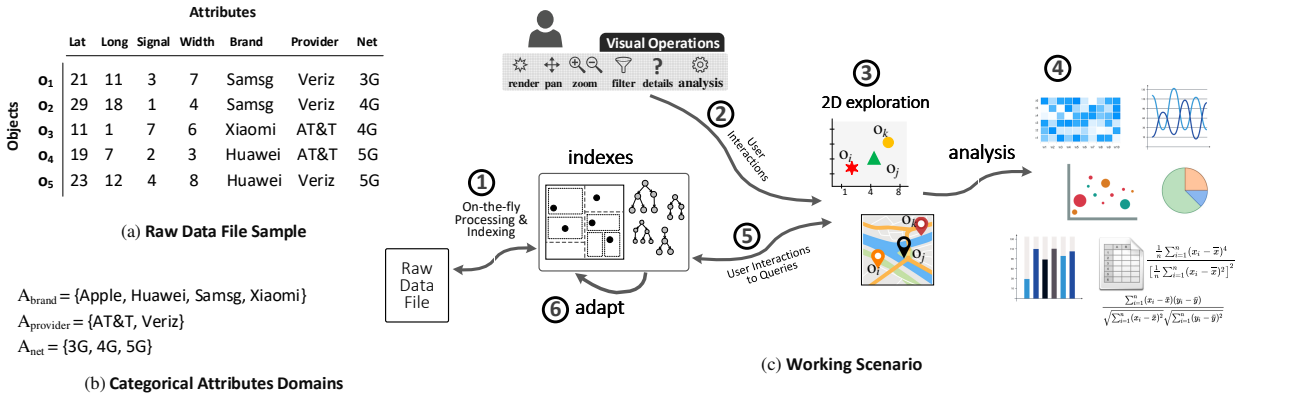


Fig. 1: (a) Raw Data File Sample (b) Categorical Attributes Domains (c) Working Scenario Overview

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

Problem Requirements and Challenges. As demonstrated in the example, the exploration of raw data visualized on a 2D plane coupled with analysis over categorical data, and statistics computations, is essential in many real-word scenarios. Group-by analysis is required to generate well-known visualization types, such as bar charts, heatmaps, parallel coordinates, binned scatter plots, radar chart, pies, etc. Many of these charts and interactions are largely employed in common data analysis tasks, such as feature extraction, correlation, OLAP analysis, clustering, regression, and comparative analysis of spatial data [38]. Beyond the visual analytics requiring group-by operations, filter operations over categorical attributes, enables the support of *effective exploration mechanisms*, such as *faceted search*. These types of analysis and queries have been widely optimized in traditional data warehouse systems, via spatial and multidimensional indexes. However, these methods require loading the data and tuning the indexes to often varying workloads.

In-situ techniques, on the contrary, attempt to avoid the overhead of moving, loading and indexing the data in a DBMS, and improve performance by progressively adapting an index as the user explores data. The key objective is how to offer fast user interactions without a preprocessing phase, i.e., low response times to visual operations performed over raw data for which no indexing and data reorganization can be applied a priori. In what follows, we highlight the requirements and technical challenges we address in this work.

In in-situ scenarios the exploratory and analysis operations are *directly evaluated over the raw file*. Using an index will enable the efficient evaluation of these operations. This index should reside in memory and be constructed *on-the-fly* coupled with a *small data-to-analysis time*, i.e., the time to parse and create the index should be kept small even for very large datasets. In this respect, the challenge is *how do we construct an index on-the-fly (small construction time) which can enable efficient query evaluation in interactive scenarios (fast response time)*? Since, the *cost of I/O operations* has a large impact on response time, the challenge is to *design the index such that access to the file is reduced, and efficient file parsing is achieved*.

Next, in the cases where *commodity hardware* is used, the data in a large raw file, as well as its index, do not fit in memory. Especially, in the cases where categorical attributes are involved, the memory requirements needed to index the dataset becomes prohibitive even for a small number of attributes. For example, Figure 2 shows the memory allocated for the initial "crude" version of our index, over different number of categorical attributes, on a relative small dataset with 100M objects (SYNTH10 dataset, Sect.8). We can observe that for 4 categorical attributes, the size of the index is 31GB, while indexing 5 attributes requires more than 64G of memory. These amounts of memory usually are not available in commodity hardware-based scenarios. The challenge here is *what part of the data we choose to index and how do we optimize the structure of the index given a predefined size of memory*?

Moreover, a challenge is related to *efficient query evaluation* over the index in exploration scenarios. The use of the information inferred by the user interactions and analysis tasks, allows the improvement of the index structure, and the selection of metadata (e.g., statistics, aggregates) which can be used to improve the evaluation performance. Thus, the challenge here is *how do we progressively adapt the index and enrich it with metadata, such that queries are efficiently evaluated*?

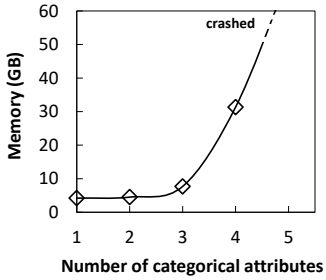


Fig. 2: Indexing Memory Requirements vs. Number of Categorical Attributes

Adaptive Indexing and In-situ Data Management. In the last decade, there are several *adaptive indexing techniques*, which aim to incrementally adjust the indexes and/or refine the physical order of data, during query processing, following the characteristics of the workload [27,22,52,25,40, 22,23,39,43]. In most cases the data has to be previously loaded/indexed in the system/memory, i.e., a preprocessing phase is considered, and such methods try to adjust the (physical) order of data, performing extensive data duplication and large memory consumption.

On the contrary, the *in-situ paradigm* has been recently adopted when analysis should be performed directly on raw data files (e.g., CSV, JSON), avoiding the overhead of fully loading and indexing the data in a DBMS. Similarly, to traditional in-database adaptive methods, in-situ techniques achieve performance by building indexes on-the-fly and progressively readjusting them as the user explores data. Works in this area have proposed techniques for progressive loading and/or indexing of raw data, for "generic" in-situ query processing (mainly range queries) [7,24,48,31,30,41], and for 2D visual operations over numeric attributes [12]. Nevertheless, in in-situ scenarios, no attention has been given for the support of: (1) exploratory aggregate queries (i.e., queries that include categorical-based group-by and filter operations) in the presence of (2) limited memory resources, such that index parameters (e.g., structure, size) are optimized to the available memory allocated for the analysis tasks.

Working Scenario. Figure 1(c) outlines our scenario. Assume that a scientist wishes to *visually explore* data using a 2D visualization technique, e.g., scatter plot, map; and *analyze* it using *visual analytics and statistics*. ① The user first selects the input file and a map as the underlying visualization layout. The file is parsed on-the-fly and an initial "crude" version of the index is constructed. ② Then, the user interacts and performs visual and analytic operations on the map. ③ For example, the user $renders$ on the map the objects located in a specific geographic area, views $details$ for the points visualized, or $filters$ out the ones that refer to a specific value of an attribute. Next, the user $moves$ (e.g., pan) the visualized region in order to explore a nearby area; or $zooms\text{-}in/out$ to explore a part of the region or a larger area, respectively. ④ Also, the user may wish to analyze the (visualized) data using visual (e.g., generate bar and line charts, heatmaps, parallel coordinates, scatter plots), or statistical methods (e.g., compute Pearson correlation or covariance). Eventually, each user interaction and analytical operation is mapped to a query evaluated over the index ⑤, and triggers the readjustment of the index structure and the update of its contents ⑥.

Our Approach. In this work, we present an innovative *indexing scheme* (VETI) and *adaptive* query evaluation techniques in the context of in-situ visual analytics. Our methods support efficient categorical-based *group-by* and *filter* operations, combined with 2D *visual interactions*, such as exploration of data points on maps or scatter plots, and statistics. VETI is built on top of a *tile-based structure* which offers efficient visual exploration over the 2D plane, enhanced with a *tree-based structure* that organizes a tile's objects based on its categorical values. The index resides in main memory and is constructed on-the-fly given the first user query.

For reducing the memory footprint during initialization, we define a *resource-aware index initialization approach*, which we formulate as an optimization problem, referred to as SIN problem. Given the amount of available memory, the initialization approach selects the initial index characteristics (e.g., which categorical attributes to be indexed in each tile), so that the memory allocated for indexing is lower than the available amount. We show that SIN is NP-hard, even in highly restricted instances. To cope with the hardness of the SIN problem, we design two efficient approximation algorithms.

We further propose a query evaluation mechanism that minimizes response time by: (1) *progressively adjusting and enriching the structure and metadata kept on the index* to the user's exploration areas and type of analysis; (2) using *the index metadata for avoiding I/Os and performing memory-based computations*; and (3) accessing the the raw file *in a sequential manner*.

In our experiments we illustrate that our approach, in most queries, reports *interactive query response times* (*less than* 0.04sec), over large raw files (e.g., 45GB). Compared to the best existing solutions, our approach *is more than 100× faster* and performs up to 2 orders of magnitude fewer I/O operations.

The proposed methods are integrated into RawVis [35], *an open source data visualization system* for in-situ visual exploration and analytics over big raw data. The source code is available under GNU/GPL.[2]

Contributions. The main contributions of this work are summarized as follows:

- We introduce a hybrid main-memory indexing scheme for raw data that organizes the objects based on spatial/numeric, as well as categorical attribute values. The index is enriched with metadata (e.g., aggregated values) for the provision of statistics.
- We formulate exploratory and analytical operations over categorical attributes, that are mapped to query operators over the underlying indexing scheme.
- We implement a resources-aware index initialization approach, based on which the index characteristics are determined based on predefined memory resources.

[2] The source code is available at: github.com/VisualFacts/RawVis

– We formulate the resources-aware index initialization as an optimization problem, and we show that even in highly restricted settings the problem is NP-hard. Also, we propose two approximation algorithms to solve it.
– We design interaction-based adaptation techniques that progressively adjust the index structure and metadata based on the user interaction.
– We implement the presented indexing scheme and the methods in an open source visualization system.[2]
– We evaluate the performance and the effectiveness of our methods using real and synthetic datasets.

Comparison to Previous Work. The basic components of the indexing scheme have been briefly presented in a preliminary version of this work [36]. Here, we significantly extend [36] as follows. (1) Regarding the tree structure (Sect. 3), we: *formally define and analyze the complexity* of the tree operations; and introduce *new operations* related to tree adaptation; (2) We introduce and study a *resource-aware index initialization* mechanism (Sect. 5), and we design *two approximation algorithms* for implementing it (Sect. 6); (3) Regarding the *query processing and index adaptation*, we provide new methods which incorporate the index initialization approach (Sect. 7); (4) In Section 8 we conduct extensive experiments with respect to [36] with several new metrics and parameters; including experiments for the two VETI initialization algorithms and using one more real dataset. Compared to our previous work in the context of in-situ visual exploration [12], we enable the support of categorical attributes and queries, which were not previously considered. In this context we: (1) define a new indexing scheme; and (2) implement categorical-based: exploratory and analytic operations, index initialization method, and query processing and index adaptation algorithms.

Outline. The paper is organized as follows. In Section 2, we present our exploration model. In Section 3, we describe the tree structure, and in Section 4 the proposed indexing scheme. In Section 5, we define the resource-aware initialization problem, and in Section 6, the proposed algorithms. Then, Section 7 presents the query evaluation and the adaptation techniques. Section 8 presents the experimental evaluation, and Section 9 the related work. Finally, Section 10 concludes the paper.

2 Exploration Model

In this section, we present the exploration model, which defines a set of exploratory and analytic operations and their translation to data-access operations. In this work, we extend the model presented in [12] by defining three new operations, i.e., grouping, filtering, and aggregating, over categorical attributes. Basic concepts of this work and notations are summarized in Table 1.

Table 1: Common Notation

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

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

Objects. Each object o_i is defined as a sorted list of d attribute values $o_i = (a_{i,1}, a_{i,2}, ..., a_{i,d})$, and associated with an *offset* f_i (a hex value) pointing to the "position" of its first attribute from the beginning of the file $\mathcal{F}$. Also, the value $a_{i,A}$ denotes the value of the attribute A for the object o_i.

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

User Interactions. The *exploration model* denotes a series of user interactions which are formulated as a set of operations (e.g., render, zoom, filter). Given a raw data file, the users arbitrarily select two numeric attributes $A_x, A_y \in \mathcal{A}$, that are mapped to the X and Y axis of a 2D visualization layout (e.g., scatter plot, map). The A_x and A_y attributes are denoted as *axis attributes*, while the rest as *non-axis*.[3]

The users visualize a rectangular area $\Phi = (I_x, I_y)$, 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 the set of *visible objects* $\mathcal{O}_\Phi \subseteq \mathcal{O}$, for which the values of their axis attributes fall within the ranges of that area.

[3] We assume that the users are familiar with the schema, the min/max values and the domains of the attributes in the data file; otherwise, they can have a preview of it, in terms of loading a small sample or parsing the file once.

In this setting, the following *operations/interactions* are defined: (1) *render*: visualizes the objects contained in the visualized area.; (2) *move*: changes the boundaries of the visualized area, i.e., a pan operation; (3) *zoom in/out* : zooms the boundaries of the visualized area keeping the center point inside Φ fixed; (4) *filter*: excludes objects visualized in Φ, based on conditions over the non-axis attributes; (5) *detail*: presents information (e.g., attributes values) related to the non-axis attributes; (6) *group*: finds group of objects based on one or more categorical attributes, i.e., similar to the group-by operation defined in SQL; (7) *analyze*: computes aggregate or statistical functions over all objects or groups of objects in the visualized area.

These operations may be combined in a sequence, e.g., render a region, filter the presented objects, group the objects based on an attribute and finally compute an average value for the groups. Thus, *a user exploration scenario is a finite sequence of operations applied by the user*.

Exploratory Query. Considering the aforementioned user operations, we proceed with defining them as data-access operators, which operate on the underlying data file. Data-access operators are essentially the building blocks of a single query applied to the data, which is referred to as *exploratory query*.

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

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

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

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

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

– *Analysis clause* L: defines two sets of algebraic aggregate functions (e.g., count, mean) [21], where each of them is applied over a set of numeric attributes, returning a single numeric value. Particularly, the *Analysis clause* defines two sets of functions: (1) L_Q that are computed over the *objects $\mathcal{O}_Q$ returned by the query*; and (2) L_G that are computed over *each group of objects* resulted by the group-by operations. Thus, the analysis clause is defined as: $\mathsf{L} = (L_Q, L_\mathsf{G})$. Note that, the support of algebraic aggregate functions in our model, enables the computation of a large number of complex statistics, e.g., Pearson correlation, covariance.[4]

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

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

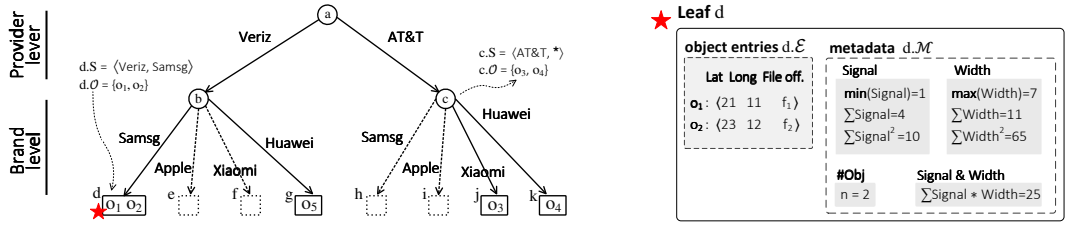
Mapping User Operations to Exploratory Queries. A user operation can be mapped to clauses of an exploratory query. Particularly, the $render$, $move$, and $zoom$ operations are implemented by the *Selection clause*; the *render* operation sets the *Selection* intervals I_x and I_y equal to the region of the visualized area, *move* sets the intervals equal to the new intervals of the shifted area and *zoom in/out* operations set the *Selection* intervals to the new coordinates of the contained/containing visualized regions, respectively. Finally, the *filter*, *details*, *group*, and *analyze* operations are implemented by the query's *Filter*, *Details*, *Group-by* and *Analysis* clauses, respectively.

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

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

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

[4] More than 90% and 75% of the statistics supported by SciPy [4] and Wolfram [5], respectively, are defined as algebraic aggregate functions [50].



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

Fig. 3: CET Tree Overview

3 CET Tree: An Index for Categorical Attributes

In this section, we present a *tree structure that organizes objects based on their categorical attribute* values, named CET (Categorical Exploration Tree).[5] CET is designed as a *lightweight*, *memory-oriented*, *trie-like* tree structure. In a nutshell, each tree level corresponds to a different categorical attribute, and edges to attribute values. Based on the tree hierarchy, each node is associated with a set of objects, that are determined based on the node path. These objects are stored in the leaf nodes.

Overall, the design of the CET tree relies on the following principles and challenges. First, considering the number of attribute value combinations which are required for categorical indexing, a significant amount of memory is required (Fig. 2). Hence, the design of a *memory-efficient categorical structure* is a major challenge, especially in our scenario, where we consider limited available resources. To reduce the memory footprint of the tree, we implement the following techniques: (1) *Each object allocates three numeric values*: (a) two numeric values for the axis attributes; and (b) one numeric value (i.e., file offset) that offers object-based, precise "connection" between object and raw file (2) *Statistics are stored only in one tree level* (in leaves), while the hierarchical structure of CET allows the efficient computation of statistics over different levels, by performing efficient, in-memory aggregate operations. (3) *The number of tree elements is reduced* (i.e., nodes/edges) during tree construction, by considering attribute characteristics, i.e., size of the attributes' domain (see Sect. 3.1).

A second challenge is to *reduce the cost of* I/O *operations* which are crucial in such I/O-sensitive settings. Exploiting the way CET stores the objects during the initialization phase (Sect. 4.3), we are able to *access the raw file in a sequential manner*. The sequential file scan increases the number of I/Os over contiguous disk blocks and improves the utilization of the look-ahead disk cache. Note that, in our experiments, *the sequential access results in about* $8\times$ *faster* I/O *operations* (more details in Sect. 7.1).

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

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

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

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

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

Synopsis Metadata. Apart from object entries, each leaf node n is associated with a set of *synopsis metadata* $n.\mathcal{M}$, which are (numeric) values calculated by algebraic aggregate functions [21] over one or more attributes of the $n.\mathcal{E}$ objects.[6] Combining the algebraic aggregate functions allows us to support a large number of statistics, e.g., Pearson correlation, covariance.[4] For example, we employ functions like *sum*, *mean*, *sum of squares of deltas* over the objects of a leaf. Using leaf metadata, we are able to compute the metadata of any internal nodes n, by aggregating the metadata of the descendant nodes of n, in a bottom-up fashion.

[5] CET is also referred to as *Tree*.

[6] *Synopsis metadata* is also referred to as *Metadata*.

Example 1. [**CET Tree**] Figure 3a presents the CET index constructed for the categorical attributes $\mathcal{C} = \{A_{Provider}, A_{Brand}\}$. The dotted lines indicate parts of the tree that will not be constructed for the particular dataset.

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

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

Regarding the *leaf nodes*, the leaf d stores the object entries $d.\mathcal{E}$ and the metadata $d.\mathcal{M}$ for the objects $d.\mathcal{O} = \{o_1, o_2\}$ that matches its values $d.\mathcal{S} = \langle$Veriz, Samsg$\rangle$ (Fig. 3b). Here, metadata here stores statistics regarding the *Signal* and the *Width* numeric attributes.

3.1 CET Operations & Analysis

This section presents the basic operations and analyzes the computational and space complexities.

Insert & Tree Construction. Procedure 1 describes the *insertion* of an object o into a CET tree h. The procedure takes as input, a tree h, the object o, and an ordered set of categorical attributes $\mathcal{C} = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$ based on which the tree organizes its objects. The tree *construction* is implemented via sequential $insert$ *operations* of its objects.

The procedure takes as input a tree h, its ordered set of attributes $\mathcal{C}$, and the object o to be inserted. Initially, the procedure constructs an empty tree (*lines* 1-3), then starts traversing the tree in a top-down fashion (*loop in line* 5). The path followed is defined by the values A_{C_i} of the object in the categorical attributes $h.\mathcal{C}$, where the values correspond to the path's edges (*line* 6). During the traversal, if the edge e_i of the path does not exist, the edge e_i and a node cn_i are constructed and inserted into the tree (*lines* 8 & 9).

After traversing the path, the leaf node l where o belongs to, is reached (*line* 11). Then, the object entry e of o is constructed and inserted to leaf entries (*line* 12). Finally, the leaf's metadata is updated w.r.t. to o (*line* 13).

Computation Complexity. Considering that he entry construction is performed in constant time $O(1)$, and the update metadata in $O(m)$ where m is the number of metadata computed in the leaves (i.e., number of aggregate values to compute), the computation complexity of the $insert$ operation is $O(m\ |\mathcal{C}|)$. Moreover, the computation complexity of the tree *construction* considering n objects is $O(n\ m\ |\mathcal{C}|)$.

Get Leaves/Objects Based on Filter Conditions. Procedure 2 implements the *get leaves operation*. Procedure returns the leaf nodes $\mathcal{L}$ of a tree h, that are matched based on the conditions defined in the Filter clause F of a query. Based on the filter conditions, the operation constructs paths

Procedure 1: insertToTree(h,$\mathcal{C}$, o)

Input: h: CET tree; $\mathcal{C}$: categorical attributes of h; o: object
Output: h: updated/cosntructed tree including o

1 **if** $h = \varnothing$ **then** //tree does not exist, construct an empty tree
2 $h.root \leftarrow$ construct root node for h
3 $h.\mathcal{C} \leftarrow \mathcal{C}$
4 $n_i \leftarrow h.root$ //initialize node n_i to root
5 **foreach** $A_{C_i} \in h.\mathcal{C}$ **do** //bottom-down traversal
6 $e_i \leftarrow$ the value of the attribute A_{C_i} in the object o //edge e_i
7 $cn_i \leftarrow$ getChild(n_i, e_i)
8 **if** $cn_i = \varnothing$ **then**
9 construct and append to node n_i: the edge e_i and a child node cn_i
10 $n_i \leftarrow cn_i$
11 $l \leftarrow n_i$ //the leaf node l for the object o
12 construct the object entry e for o and insert e into leaf entries $l.\mathcal{E}$
13 update l metadata w.r.t. o
14 **return** h

Procedure 2: getLeavesBasedOnFilter(h, F)

Input: h: CET tree; F: Filter clause
Output: $\mathcal{L}$: leaf nodes selected based on the categorical filter conditions of F

1 $p \leftarrow \varnothing$ //path expression example: / a/ * / (c || d)
2 **foreach** $c \in h.\mathcal{C}$ **do** //generate paths based on filter conditions
3 **if** F includes condition in attribute c **then**
4 append "/ *filter conditions for the attribute c*" to p
5 **else**
6 append "/ $*$" to p
7 $\mathcal{L} \leftarrow$ getLeavesByPaths(h, p)
8 **return** $\mathcal{L}$

p starting from the root to the leaf nodes (*loop in line* 2), and the getLeavesByPaths (*line* 7) returns the leaves $\mathcal{L}$ reached by all paths. The *get objects based on filter conditions* operation (not shown for simplicity) is implemented by returning the object's entries of the leaves $\mathcal{L}$.

Computation Complexity. The worst case corresponds when we have to access all leaf nodes in a tree. Let ℓ be the number of leaves, the computation complexity of $get\ leaves\ based\ on\ filter\ conditions$, for a tree that contains $\mathcal{C}$ attributes is $O(\ell|C|)$. The value of the number ℓ is defined in the next paragraphs.

The operation $get\ objects\ based\ on\ filter\ conditions$ is implemented by inserting the objects of each leaf returned by the previous operation, in a linked list. The cost of the insertion in a linked list is $O(1)$; hence, the complexity is $O(\ell\ |C|\ |h.\mathcal{O}|)$.

Expand Tree Based on New Attributes. The *expand tree* operation, expand$(\mathcal{L}, \mathcal{C})$, enriches a tree by including new categorical attributes and reorganizes the objects based on these attributes. The operation takes as input a set with the new categorical attributes $\mathcal{C}$, and a subset of leaf nodes $\mathcal{L}$

of a tree h, which should be reorganized based on $\mathcal{C}$. Note that, the leaf nodes $\mathcal{L}$ could correspond to the entire set of leaves in the tree h. For each leaf node $l_i \in \mathcal{L}$, a subtree h_i having l_i as root is constructed, where h_i has one level for each attribute $A_C \in \mathcal{C}$ and leaf nodes $\mathcal{L}_{h_i}$. The objects of each leaf node l_i are organized based on $\mathcal{C}$ attributes and stored to the leaf nodes $\mathcal{L}_{h_i}$ of the generated tree h_i. Further, the metadata of the new leaf nodes $\mathcal{L}_{h_i}$ are computed.

The *expand* operation is used during the user exploration, where a query request attributes not existing in the tree. In such cases, the values of the missing attributes retrieved from the file expand the tree and reorganize the objects (see Sect. 7).

Computation Complexity. The wort case is when the leafs $\mathcal{L}$ you have to expand, enclose all the tree objects $h.\mathcal{O}$. In such case, you have to construct a tree for each leaf $l_i \in \mathcal{L}$, organizing its objects $l_i.\mathcal{O}$ based on $\mathcal{C}$. Recall that the complexity of constructing a tree with n objects, $\mathcal{C}$ attributes, and computing m number of aggregate values (metadata) is $O(m\ n\ |\mathcal{C}|)$. Hence, in our case, the complexity of constructing the trees for the $\mathcal{L}$ leaves is: $O(m\ |l_0.\mathcal{O}|\ |\mathcal{C}|) + O(m|l_1.\mathcal{O}||\mathcal{C}|)+\cdots+O(m|l_{|\mathcal{L}|-1}.\mathcal{O}||\mathcal{C}|)$, with $\sum_{i=0}^{|\mathcal{L}|-1} |l_i.\mathcal{O}| = |h.\mathcal{O}|$. Therefore, the worst case complexity for the expand operation is $O(m\ |h.\mathcal{O}|\ |\mathcal{C}|)$.

Tree Space Complexity Analysis. Considering the CET insertion process, a node n is included in the tree, only if its sequence of values $n.S$ is associated with one of its objects. In other words, nodes that represent combinations of attribute values that do not appear in the data objects are not inserted in the tree structure. As a result, the number of nodes depends on the distinct values of the attributes in the objects included in the tree.

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

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

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

Since the memory allocated by each node is almost the same (except for the leaf nodes where metadata is stored), here, for simplicity, we assume that all nodes allocate equal memory size. Furthermore, all object entries have the same size (about four numeric values).

Therefore, the *space complexity* of CET is: $O(\alpha + \alpha \sum_{i=1}^{|h.\mathcal{C}|} \Gamma_i + \beta\,|h.\mathcal{O}|)$, where α and β are the memory allocated by a node and an object entry, respectively.

Based on the formulas above, we can easily verify that the number of nodes in a tree h depends on the mapping of its attributes $h.\mathcal{C}$ to the levels of the tree and the size of their domain. Assuming that the data follow a uniform distribution over the domain values of each attribute, we can reduce the number of nodes (and edges) in the tree, by placing the attributes at the levels of the tree in a top-down way based on their domain size, i.e., smaller domains are placed closer to the root. So, *constructing a tree following this attribute order, will result in lower space requirements*. In our experiments, this attribute order led to up to 10% reduction in total index memory requirements, compared to a random order (Fig. 15).

4 VETI: A Tile-Tree Adaptive Index

In this section, we present the VETI indexing scheme (Visual Exploration Tile-Tree Index), that combines the tile-based index presented in [12] and the CET tree structure. The design of VETI relies on the basic challenges posed by the in-situ exploration scenarios. First, the *index construction should entail a small overhead in the raw data-to-analysis time*. To this end, a lightweight, "crude" version of VETI is initially constructed on-the-fly, by parsing the raw file once. Moreover, the characteristics of this initial VETI version are defined by considering query and data-related factors in order to improve the performance of the initial user interactions. Second, during the exploration, the *index should support efficient exploratory and analytic operations*. Thus, based on user exploration, efficient structure adaptation and object reorganization techniques are employed to adjust the index to user interactions. Third, considering the *limited available resources*, VETI uses lightweight tree and tile structures with predefined memory resources allocated to them (Sect. 5).

4.1 Tile-based Structure

Our work is built on top of the VALINOR tile-based index, referred also as *tile-structure* [12], for the support of exploration operations over a 2D representation of raw data. VALINOR is a *tile-based multilevel* index, which is stored in memory and *organizes the data objects of a raw file into hierarchies of non-overlapping rectangle tiles*. Each tile is constructed over a range of values for the A_x and A_y axis attributes.The index is initialized with a number of tiles and progressively adjusts itself to the user interactions, by splitting these tiles into more fine-grained ones, thus forming a hierarchy of tiles. Next, we summarize the basic concepts of the index.

A *tile* t is a part of the Euclidean space defined by two left-closed, right-open intervals $t.I_x$ and $t.I_y$, and $\mathcal{T}$ is the set of tiles defined in the tile-structure.

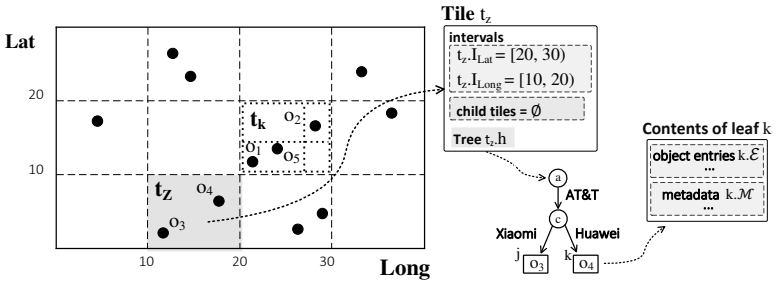


Fig. 4: VETI Index Overview

In this work, we assume *hierarchies of tiles* (i.e., forest), although a hierarchy with a single root tile can also be defined. In each level of the hierarchy, there are no overlaps between the tiles of the same level, i.e., disjoint tiles.

A *non-leaf tile* t can have an arbitrary number of *child tiles*, whereas *leaf tiles* are the tiles without children and can appear at different levels in the hierarchy. Further, a *non-leaf tile* covers an area that encloses the area represented by any of its children. That is, given a non-leaf tile t defined by the intervals $t.I_x = [x_1, x_2)$ and $t.I_y = [y_1, y_2)$; for each child tile t' of t, with $t'.I_x = [x'_1, x'_2)$ and $t'.I_y = [y'_1, y'_2)$, it holds that $x_1 \leq x'_1, x_2 \geq x'_2, y_1 \leq y'_1$ and $y_2 \geq y'_2$.

Each tile t *encloses a set of objects* $t.\mathcal{O}$, where the axis values $a_{i,x}$ and $a_{i,y}$ of each object $o_i \in t.\mathcal{O}$ fall within the intervals of the tile t, $t.I_x$ and $t.I_y$, respectively.

4.2 VETI: Combining Tiles and Trees

The VETI index (Visual Exploration Tile-Tree Index) combines the VALINOR tile-based index with the CET tree.

Given a *raw data file* $\mathcal{F}$, two *axis attributes* A_x and A_y, and a set $\mathcal{C}$ of *categorical attributes* of the objects stored in $\mathcal{F}$, the VETI index $\mathbb{I}$ organizes the objects stored in $\mathcal{F}$ into hierarchies of non-overlapping tiles based on its A_x, A_y values, where tiles are also associated with CET trees which organize objects based on categorical attributes from $\mathcal{C}$.

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

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

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

The basic operations of the VETI are: *initialization* (Sect. 4.3), *query evaluation* (Sect. 7), and *adaptation* (Sect. 7.2).

Example 2. [**VETI Index**] Figure 4 presents the VETI index of the example data in Fig. 1. VETI divides the 2D space into 4×3 equally sized disjoint tiles, and a tile t_k is further divided into 2×2 subtitles of arbitrary sizes, forming a tile hierarchy.

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

4.3 VETI Initialization

In our approach, we do not require any preprocessing phase. The index is constructed on-the-fly when the user first requests to visualize the file. During the initialization phase, the following tasks are realized. Initially, the characteristics of the index are determined; then, the file is parsed and the objects populate the index; finally the query is evaluated.

Algorithm 1 outlines the initialization phase. It takes as input, the raw file $\mathcal{F}$, the axis and categorical attributes A_x, A_y and $\mathcal{A}_\mathcal{C}$, the memory budget $\mathcal{B}$ that is available for the index, and the first exploratory query Q_0; and returns the initialized index $\mathbb{I}$ and the results $\mathcal{R}_0$ of the Q_0.

Initially, the categorical attributes are sorted by their domain size (*line* 1), in order to reduce the size of the constructed trees (Sect. 3.1). Then, considering query evaluation performance, the tile structure characteristics are determined (i.e., number, size and intervals of the tiles) and the tiles $\mathbb{I}_\mathcal{T}$ are constructed (*line* 2). Next, based on the constructed tile structure and memory budget $\mathcal{B}$, the assignments $\mathbb{I}_\Pi$ of trees to tiles $\mathbb{I}_\mathcal{T}$ are determined (*line* 3). Details about the assignment selection are presented in the next section (Sect. 5).

In the next part (*loop in line* 4), the algorithm scans the file $\mathcal{F}$ and reads, for each object o_i, the values of the axis attributes $a_{i,x}$, $a_{i,y}$, the categorical attributes, and the attributes which are required to evaluate the Analysis clause of Q_0 (*line* 5). Next, the tile t_i that encloses o_i is found

Algorithm 1. Initialization ($\mathcal{F}$, A_x, A_y, $\mathcal{C}$, Q_0, $\mathcal{B}$)

Input: $\mathcal{F}$: raw data file; A_x, A_y: axis attributes; $\mathcal{C}$: categorical attributes; Q_0: first query; $\mathcal{B}$: memory budget
Output: $\mathbb{I}$: initialized index; $\mathcal{R}_0$: result of query Q_0

1 sortAttributes($\mathcal{C}$) //sort attributes based on the domain size -in order to reduce tree size-
2 $\mathbb{I}_{\mathcal{T}} \leftarrow$ IT.constructTiles($A_x, A_y, \mathcal{A}_{\mathcal{C}}, Q_0$) //determine the number, size & intervals of the tiles, and construct them
3 $\mathbb{I}_{II} \leftarrow$ find tile-tree assignments //see Sect. 5
4 **foreach** $o_i \in \mathcal{F}$ **do** //read objects from file, insert them to trees & evaluate Q_0
5 *read* from $\mathcal{F}$ the values of axis and categorical $\mathcal{C}$, and the attributes required to evaluate the Q_0 Analysis clause
6 use the o_i attributes to evaluate Q_0
7 $t_i \leftarrow$ find the tile $t_i \in \mathbb{I}_{\mathcal{T}}$ that encloses o_i based on its axis attributes values
8 insertToTree($t_i.h, \mathcal{C}, o_i$) //insert o_i to tree $t_i.h$ (Proc. 1, Sect. 3.1)
9 **return** $\mathbb{I}$, $\mathcal{R}_0$

(*line* 7), and the insertToTree method (Procedure 1), inserts o_i into the tree $t_i.h$ (*line* 8). During the insertion, the object entry is constructed, the tree metadata are updated, and new parts (i.e., nodes, edges) of the tree may be constructed.

Tile Structure Initialization. The constructTiles method is defined by the tile initialization strategy IT, and determines the tile structure characteristics, e.g., number, size and intervals of the tiles. These can be defined via numerous approaches [44]. For instance, they can be either given explicitly by the user, e.g., in a map the user defines a default scale of coordinates for the initial visualization; determined by the visualization setting, considering certain characteristics (e.g., visualization type, screen size/resolution), previous sessions, task, user preferences [44,28,8,47]; or computed based on techniques that consider data characteristics in order to divide the data space into equal size tiles [11].

In this work, we use a *locality-based initialization* approach, adapted from [12], which considers: (1) the user exploration entry point, i.e., the position of the first user query in the 2D space; (2) the window size of the first query; and (3) the locality-based behavior of the exploration scenarios, i.e., users are more likely to explore nearby regions of their initial entry point [53,29,47,8,51,16]. It follows a probability distribution (bivariate normal distribution) for defining the number and size of the tiles, with a tile structure that is more fine-grained (i.e., having a larger number of smaller tiles) in the area around the initial query, whereas the size of tiles becomes larger as their distance from the initial query increase. Increasing the number (i.e., decreasing the size) of tiles near the first query, increases the probability that subsequent user queries in this neighborhood overlap with fully-contained tiles, which in turn reduce the number of I/O operations.

As an *I/O operation* we denote the file access in order to read an object (i.e., all attributes values), or some of its attribute values. So, the number of I/O operation corresponds to the number of rows (i.e., objects) from which we read attributes' values.

Particularly, during query evaluation, the metadata stored in the tree of each tile can be used to reduce the number of I/O's. However, the use of metadata depends on whether an overlapping tile is fully or partially-contained in the window query. In case of fully-contained tiles, tiles' metadata can be used to reduces the number of I/O's. On the other hand, in the partially-contained case, in order to compute the metadata, we have to access the file and read the values for the objects contained in the partial section. More details about query evaluation are presented in Section 7.

5 Resource-aware Index Initialization

In this section, we present the initialization of the CET trees and their assignment to tiles. Recall from Figure 2, that more than 64GB is required for VETI to create full trees from five categorical attributes. Our goal here is to determine the structures of the trees (the categorical attributes that will be placed as levels in the tree) and assign them to tiles based on the "utility" of each tree. The latter depends on the utility of the categorical attributes it contains; we consider that an attribute has a higher utility score when its inclusion in the tile's tree is expected to improve *the performance in the user exploration scenario*.

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

5.1 Preliminaries

Before we formally introduce our problem, we present some necessary definitions. [7]

Tile Utility. Let a tile t, the *tile utility* $\rho_t \in [0,1]$ formulates the possibility that a future exploratory query will overlap with t. For the distribution of ρ_t, we follow the approach in [12]. Based on the locality-based characteristics of 2D exploration scenarios, users are more likely to explore nearby regions of their initial exploration entry point [53,29,47,8,51,16]. Thus, given an initial query Q_0, the next queries are more likely to overlap with tiles near Q_0 and the value of ρ_t is larger in tiles near Q_0. Particularly, [12] models the probability that a query overlaps with a tile t, considering a bivariate normal distribution based on the distance of the tiles from the center of Q_0.

Attribute Score & Tree Utility. We assume that *each categorical attribute c has a score $c.S \in [0,1]$*, that represents the probability that a future query will request this attribute. We define the attribute score based on the "repetitive calculation of statistics" that appears in exploration scenarios [50], i.e., we assume that the attributes requested by the initial query Q_0, are more likely to be requested by next user interactions. Using the attributes scores, the tree utility is defined as follows.

Given a tree h, the *tree utility* $\rho_h \in [0,1]$ formulates the possibility that an exploratory query requests information stored in the tree. Without loss of generality, we define

[7] Note that, several of the problem's involved metrics (e.g., tile and tree utility) can be computed using a large number of factors, e.g., device and visualization type, interface, user profile, task, domain [44]. However, this is beyond the scope of this work.

Table 2: SIN Example: Tile-tree Assignment Utilities

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

the *tree utility* ρ_h as the normalized sum of the scores of the tree attributes $h.\mathcal{C}$:

$$\rho_h = \frac{\sum\limits_{\forall c \in h.\mathcal{C}} c.S}{\sum\limits_{\forall c \in \mathcal{C}} c.S} \tag{1}$$

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

Regarding the *categorical attribute scores*, the attributes P and B are included in Q_0 and assigned with a score 0.8, whereas N has score 0.1. Additionally, assume the trees: $h_{P,B}.\mathcal{C} = \{P, B\}$, and $h_{P,N}.\mathcal{C} = \{P, N\}$. The tree $h_{P,B}$ that includes both attributes of Q_0 will have a larger utility than $h_{P,N}$ which includes only one of them. Based on the Eq. 1 the *tree utilities* are $\rho_{h_{P,B}} = 0.96$ and $\rho_{h_{P,N}} = 0.82$.

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

Tile-Tree Assignment Utility Each *tile-tree assignment π_t^h is associated with a utility* $\pi_t^h.\omega \in [0, 1]$, which formulates the possibility that a query is going to request information from the tile t involving the attributes $h.\mathcal{C}$ of its tree. Intuitively, an assignment's utility formulates the "effectiveness of the information" contained by a tile-tree assignment during query evaluation. The *tile-tree assignment utility* is defined as the joint probability of the *tile utility* ρ_t and the *tree utility* ρ_h:

$$\pi_t^h.\omega = \rho_t \cdot \rho_h \tag{2}$$

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

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

Example 4. [***Assignments***] Consider the index of the Example 3. The *attributes-based tree powerset* for the attributes P, B, N is: $HP_{\{P,B,N\}} = \{h_{P,B,N}, h_{P,B}, h_{P,N}, h_{B,N}, h_P, h_B, h_N, h_{empty}\}$. An assignment over $\mathbb{I}$ can include any tree from this set, e.g., the *index assignment set* $\mathbb{I}_{\Pi} = \{\ \pi_{t_1}^{h_{P,B,N}}, \pi_{t_2}^{h_N}, \pi_{t_3}^{h_{P,B,N}}\ \}$ assigns the tree $h_{P,B,N}$ to tiles t_1, t_3; h_N to t_2, and no assignments (i.e., empty tree) are made for tiles t_4, t_5, t_6.

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

$$\Omega(\mathbb{I}_{\Pi}) = \sum_{\forall \pi_t^h \in \mathbb{I}_{\Pi}} \pi_t^h.\omega \tag{3}$$

Index Initialization Cost. The *index initialization cost* $\mathbb{I}_{cost}$ denotes the resources (e.g., memory, time) that are required for the VETI initialization. Here, as *resource* we only refer to memory. Specifically, the index initialization cost denotes the memory allocated by the index structures (i.e., tiles, trees, metadata), and does not include the memory required by the object entries that allocate a constant amount of memory; each object allocates three numeric values (Sect. 3).

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

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

5.2 Problem Definition & Analysis

The *Resource-aware Index Initialization* (SIN) problem is defined as follows.

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

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

Example 5. [***SIN Problem***] Based on Example 3, we assume the six tiles ($t_1 - t_6$) and the attributes P, B, N. Table 2 presents the tile-tree assignment utilities (Eq. 2) for all the possible assignments over the tiles, the tiles utilities (in parenthesis), and the cost of every possible tree (in parenthesis). Here, the cost of the trees is expressed in number of tree nodes, and we assume that all combinations of attribute values appear in the data (for space complexity see Sect. 3.1). For example, based on the domain of the attributes P, B, N (Example 3) the tree $h_{P,B,N}$ has cost (number of nodes) equal to 33. Also, the assignment $\pi_{t_1}^{h_{P,B,N}}$ that assigns tree $h_{P,B,N}$ to tile t_1 has utility $\pi_{t_1}^{h_{P,B,N}}.\omega = 0.6$.

In order to solve SIN from Table 2 we have to determine the tile-tree assignments that maximize the total index utility and keeps the assignment cost lower than the available budget. Let 50 be the budget available for the tree structures, expressed in total number of tree nodes in the index. We can verify, that the index assignment set $\mathbb{I}_{\Pi} = \{ \pi_{t_1}^{h_{P,B}}, \pi_{t_2}^{h_{P,B}}, \pi_{t_3}^{h_{P,B}}, \pi_{t_4}^{h_{P,B}}, \pi_{t_5}^{h_{P,B}}, \pi_{t_6}^{h_P} \}$ corresponds to a solution of SIN. Particularly, these assignments result in a total index utility $\Omega(\mathbb{I}_{\Pi})$ equal to 0.9 (which is the largest), and the cost $\mathbb{I}_{\mathcal{H}cost}$ of its trees is 47.

Next, we show that even in highly restricted instances the SIN problem is NP-hard.

Theorem 1. The SIN problem is NP-hard.

PROOF. We reduce our problem to the 0-1 Knapsack Problem (KP), which is known to be NP-hard [**?**] and which states that given a bin with *capacity* ψ; a set of *items* $\mathcal{I}$, where each item $i \in \mathcal{I}$ has a *weight* $w(i)$ and a *profit* $p(i)$; find a subset $\mathcal{Z} \subseteq \mathcal{I}$ such that the total profit $P = \sum_{\forall z \in \mathcal{Z}} p(z)$ is maximized and the total weight $W = \sum_{\forall z \in \mathcal{Z}} w(z) \leq \psi$.

In the proof we consider a *restricted instance* of SIN, where: (1) the index $\mathbb{I}$ contains only one tile t, i.e., $\mathbb{I}_{\mathcal{T}} = \{t\}$; (2) the tile utility is equal to one, i.e., $\rho_t = 1$; and (3) each attribute c is associated with a construction cost (i.e., allocated memory) $mem_{cost}(c)$ that corresponds to the memory overhead, when c is included in a tree, with $mem_{cost}(c)$ be the same in any tree. Here, the tree construction cost for a tree h, is equal to the sum of its attributes construction costs, i.e., $\sum_{\forall c \in h.\mathcal{C}} mem_{cost}(c)$.

Remark that, the latter defines to a restricted case regarding the tree allocated memory. Here, in contrast to the space complexity presented in Section 3, the constitution of an attribute in the tree's allocated memory is independent of the other attributes included in the tree. In other words, the increase in memory size of a tree (resp. decrease) resulted by adding (resp. removing) an attribute, is the same in each tree.

For the reduction, we use the following *associations between the* KP *and the* SIN: (1) the bin corresponds to the tile t; (2) bin capacity ψ corresponds to the memory budget $\mathcal{B}$ minus the cost for constructing the tiles $\mathbb{I}_{\mathcal{T}cost}$; (3) each item $i \in \mathcal{I}$ corresponds to a categorical attribute $c \in C$; (4) the profit $p(i)$ of an item i corresponds to the score $c.S$ of the attribute c; and (5) the weight $w(i)$ of an item i corresponds to the memory $mem_{cost}(c)$ allocated by the attribute c.

Based on the aforementioned association, we can verify that, given a set of attributes $\mathcal{K}$; the assignment that assigns the tree h with $h.\mathcal{C} = \mathcal{K}$ to tile t, results to index utility $\Omega(\mathbb{I}_{\Pi}) = \sum_{\forall c \in \mathcal{K}} c.S$. Hence, index utility $\Omega(\mathbb{I}_{\Pi})$ corresponds to the total profit P in the KP problem. Moreover, we can verify that if the budget constraint holds in the SIN problem, the capacity constraint also holds in the KP. Regarding the budget/capacity constraints, we have the following. In the SIN solution, $\mathbb{I}_{cost} \leq \mathcal{B} \Leftrightarrow \mathbb{I}_{\mathcal{T}cost} + \sum_{\forall c \in \mathcal{K}} mem_{cost}(c) \leq \mathcal{B}$. Based on the reduction, we can verify that the cost of constructed the trees of the index $\sum_{\forall c \in \mathcal{K}} mem_{cost}(c)$ in the SIN problem, is equal to the total weight W of the items in the KP problem; so, we have: $\mathbb{I}_{\mathcal{T}cost} + W \leq \mathcal{B} \Leftrightarrow W \leq \mathcal{B} - \mathbb{I}_{\mathcal{T}cost}$. Moreover, from the reduction we have that $\mathcal{B} - \mathbb{I}_{\mathcal{T}cost}$ corresponds to the bin capacity ψ. So, if the budget constraint holds in the SIN problem, the capacity constraint also holds in the KP. ■

6 SIN Algorithms

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

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

6.1 Preliminaries

Candidate Trees. The *candidate trees* is a subset of the $HP_{\mathcal{C}}$ set, that contains $|\mathcal{C}|$ trees with "promising" categorical attributes, i.e., the ones that are expected to increase the index utility. To determine the promising attributes, we sort the attributes $\mathcal{C}$ in a descending order, by a *gain score* $gain(c)$, that combines: (1) the *attribute score* $c.S$ (Sect. 5.1); and (2) the *attribute memory cost*. The latter formulates the memory overhead, when c is included in a tree. Since the memory cost of a tree depends on the number of distinct values of its attributes, we consider the domain size $|dom(c)|$ to quantify each attribute's memory cost. We define the *gain score* of an attribute c as: $gain(c) = \frac{c.S}{|dom(c)|}$.

Given a gain-ordered list L_g of attributes $\mathcal{C}$, the *candidate tree set* $\mathcal{H}$, is defined by $|\mathcal{C}|$ trees, where each tree

$h_i \in \mathcal{H}$ contains the first $(i+1)^{th}$ attributes of L_g. Therefore, the *candidate tree set* is $\mathcal{H} = \{h_0, ..h_{|\mathcal{C}|-1}\}$, with $h_i.\mathcal{C} = \{L_g[0], ...L_g[i]\}$.

The candidate tree set can be characterized as a *small number of trees*, where each of them has *different memory cost* (i.e., number of attributes), while containing *as many "promising" attributes as possible*.

Example 6. [***Candidate Trees***] From Example 3 we have the *attribute scores:* $A_P.S = 0.8$, $A_B.S = 0.8$, and $A_N.S = 0.1$. Also, we have the the following *domain sizes*: $|dom(P)| = 2$, $|dom(B)| = 4$, and $|dom(N)| = 3$. Hence, the *attributes gain scores* are $gain(P) = 0.8/2$, $gain(B) = 0.8/4$, and $gain(N) = 0.1/3$. Based on the gain scores, the sorted list of attributes is $L_g = \{P, B, N\}$. Thus, the *candidate trees* are $\mathcal{H} = \{h_P, h_{P,B}, h_{P,B,N}\}$, where $h_P.\mathcal{C} = \{P\}$, $h_{P,B}.\mathcal{C} = \{P, B\}$, and $h_{P,B,N}.\mathcal{C} = \{P, B, N\}$.

Tile-Tree Assignment Cost Estimation. The tile-tree assignment cost denotes the memory allocated by the assignment's tree. Recall from Section 3 that, a tree is populated during the initial file parsing, with the distinct values that appear in the categorical attributes of the data objects it contains. Therefore, the actual tree size is not known a priori, and should be estimated during index initialization.

As estimation we consider the worst case (i.e., the maximum memory a tree can require), that is defined by the maximum number of nodes the tree can have (see *Tree space complexity analysis* in Sect. 3.1). Let $nodes_{max}(\nu, \mathcal{C})$ denote the maximum number of nodes of a tree that contains $\mathcal{C}$ attributes and ν objects.

Assuming a uniform distribution of objects over the tiles, the estimated number of nodes per tile is $\nu_t = |\mathcal{O}_{DS}| \cdot \frac{areaSize(t)}{areaSize(DS)}$, where $areaSize(DS)$ and $areaSize(t)$ are the sizes of the 2D areas defined by the dataset objects (i.e., grid area size $|dom(A_x)| \cdot |dom(A_y)|$), and a tile t, respectively; and $|\mathcal{O}_{DS}|$ is the number of objects in the dataset. So, the *maximum cost estimation* for an assignment π_t^h is $\pi_t^h.\Phi = nodes_{max}(\nu_t, \mathcal{C}) \cdot n_{cost}$, where n_{cost} is the memory allocated by a single node.[8]

Tree Eviction. Using the estimated cost per tree, the assignments are selected so that the overall estimated memory cost for constructing the trees is lower than the budget. However, there are cases where the memory allocated for the trees during the file parsing is larger than the estimated. In such cases, trees are evicted (removed from memory) in order to satisfy the budget constraint.

To this end, we adopt a simple eviction policy, where the trees are selected to be evicted based on their tile utility value ρ_t. The tile with the lowest utility is selected, and its tree is removed from memory. When a tree gets evicted, its structure is destroyed (i.e., nodes, edges, and metadata are removed from the memory) and the object entries are reassigned to a single root node attached to the corresponding tile. Also, in case where the object entries exceed the memory budget, we use an eviction mechanism [12], which writes a tile's object entries to the disk, and fetches them from it when a future query overlaps with that tile.

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

Algorithm 2. GRD ($\mathbb{I}_\mathcal{T}, \mathcal{A}_\mathcal{C}, \mathcal{B}_\Pi$)

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

1 $\mathcal{H} \leftarrow$ generateCandTrees($\mathcal{A}_\mathcal{C}$) //generate candidate trees
2 **foreach** $(t, h) \in \mathbb{I}_\mathcal{T} \times \mathcal{H}$ **do** //generate assignments & compute utilities
3 compute $\pi_t^h.\omega$ and $\pi_t^h.\Phi$ //assignment utility (Eq.2) & appr. cost (Sect.5.1)
4 $\pi_t^h.score \leftarrow$ assgnScore($\pi_t^h.\omega$, $\pi_t^h.\Phi$) //compute assignment score w.r.t. assignment's utility $\pi_t^h.\omega$ and appr. cost $\pi_t^h.\Phi$
5 push π_t^h to W_π //initialize assignments max-heap
6 $Cost_\Pi \leftarrow 0$;
7 **while** $Cost_\Pi < \mathcal{B}_\Pi$ and $W_\pi \neq \varnothing$ **do** //select assignments
8 $\pi_{t_\gamma}^{h_\gamma} \leftarrow pop(W_\pi)$ //select (and remove) the top assignment
9 insert $\pi_{t_\gamma}^{h_\gamma}$ into $\mathbb{I}_\Pi$ //the selected assignment is inserted into assignments list
10 $Cost_\Pi \leftarrow Cost_\Pi + \pi_{t_\gamma}^{h_\gamma}.\Phi$
11 **return** $\mathbb{I}_\Pi$

6.2 Greedy Tile-Tree Assignments Algorithm (GRD)

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

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

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

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

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

Obviously, if an assignment for a tile t is selected, the rest of the assignments referring to t are not examined.

Example 7. [***GRD Algorithm***] In this example, we assume that the estimated cost $\pi_t^t.\Phi$ of an assignment (Sect. 6) is equal to the cost presented in Table 2.Also, the assignment score is equal to assignment utility presented in Table 2. Finally, as in Example 5, we assume a budget of 50.

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

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

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

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

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

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

Basic Characteristics. BINN has the following steps: (1) The *bin-based tile organization* phase, in which the tiles are grouped into bins. The tree assignments are defined at bin-level and, thus, are not "strictly" affected by tile-specific factors, which in many cases may not be accurately estimated, such as the tile probability and the tile-tree assignment cost. (2) The *assignment initialization phase*, which defines "default" assignments for (some) tiles. These assignments may be updated/replaced during the assignment selection process. Hence, this phase enables the algorithm to "impose" assignments to a set of tiles and/or "influence" the assignment selections that follow. For example, BINN may assign a tree with one attribute to the tiles with probability larger than a threshold, or to the top-k tiles. (3) The *assignment selection phase*, which traverses and assigns a tree to each bin, by considering the selected and the default assignments in the rest of the bins.

Algorithm 3. BINN ($\mathbb{I}_\mathcal{T}, \mathcal{A}_\mathcal{C}, \mathcal{B}_\Pi$)

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

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

BINN vs. GRD. BINN tackles a basic shortcoming of the GRD algorithm. Particularly, GRD allocates most of the budget assigning trees with all categorical attributes included (Fig. 10). As a result, trees are assigned to a smaller number of tiles. On the other hand, the bin-based approach adopted by the BINN algorithm leads to a more "balanced" allocation of the budget, with more tiles being assigned with trees having fewer categorical attributes. As demonstrated in our evaluation (Sect.8), in many cases, the small number of trees assigned by GRD compared to BINN has great impact in algorithms performance. In general, BINN is more than 1.5× faster and perform the half I/Os compared to GRD. To also remark that, in several cases BINN is more than 100× faster (Fig. 13).

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

In the next step, if an assignment initialization strategy AI has been defined (*line* 6), the algorithm performs the assignments initialization phase. For each bin i (*loop in line* 7), the function selectInitialTreeForBin determines the default tree $L_\mathcal{H}[i]$ of the i^{th} bin (*line* 8). More details about the functions are presented in the next paragraph. Next, the function assignmentsCostInBin computes the cost of this assignment,

considering the assigned tree $L_{\mathcal{H}}[i]$ for the tiles $L_{\mathcal{T}}[i]$ of bin i (*line* 9).

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

Strategies Details. Without loss of generality, in our experiments, the binning strategy BS uses equal frequency binning to define the bin intervals. The selectInitialTreeForBin (*line* 8) and selectTreeForBin (*line* 12) functions may consider several factors such as: the already assigned trees $L_{\mathcal{H}}$ (assigned either during initialization, or during selection), the currently available budget ($\mathcal{B}_{\Pi} - Cost_{\Pi}$), the distribution of tile probabilities, etc. In our implementation, we define a simple selectTreeForBin function, which assigns to each bin the candidate tree $\mathcal{H}[k]$ with the larger number of attributes, such that the cost of already selected and initialized assignments is lower than the budget. Hence, the function selectTreeForBin for a bin i selects: $\mathcal{H}[k]$ s.t. $\arg\max \mathcal{H}[k]$ and $Cost_{\Pi} + \text{assignmentsCostInBin}(L_{\mathcal{T}}[i], \mathcal{H}[k]) < \mathcal{B}_{\Pi}$.

Example 8. [***BINN Algorithm***] As in the previous example, we assume that the estimated assignment cost and score are equal to the cost and the utility presented in Table 2, and the budget is equal to 50. We adopt an equal frequency binning strategy to define the bin intervals (e.g., we consider two bins), and as selectTreeForBin we use the method described above. Based on the tile utilities shown (in parenthesis) in Table 2, the following bins of tiles are defined: $L_{\mathcal{T}} = \{\{t_1, t_2, t_3\}, \{t_4, t_5, t_6\}\}$.

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

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

We can easily verify that a large number of "rational" selectInitialTreeForBin (*line* 8) and selectTreeForBin (*line* 12) functions, cost $O(|L_{\mathcal{T}}[i]||\mathcal{C}|)$ in order to select a tree for bin i. In each selection, these functions examine the candidate trees CT, and in the same time compute the different costs. Since, in such functions the cost is computed during the selection, the function assignmentsCostInBin is omitted. Note that, the same complexities also hold in the functions used in our implementation (see previous paragraphs). Thus, in the worst case, the *loop in line 7* costs $O(|\mathbb{I}_{\mathcal{T}}||\mathcal{C}|)$; the same also holds for the *loop in line 11*. In the last loop in the worst case, the insert operation (*line* 16) is executed $|\mathbb{I}_{\mathcal{T}}|$ times. Thus, the cost of the insertions in the linked list is $O(|\mathbb{I}_{\mathcal{T}}|)$.

Therefore, in the the worst case the complexity of BINN is the sum of the aforementioned steps:

$O(|\mathbb{I}_{\mathcal{T}}|\ log|\mathbb{I}_{\mathcal{T}}| + |\mathbb{I}_{\mathcal{T}}| + |\mathbb{I}_{\mathcal{T}}| + |\mathcal{C}|\ log|\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}||\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}||\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}|)$
$= O(|\mathbb{I}_{\mathcal{T}}|\ log|\mathbb{I}_{\mathcal{T}}| + |\mathcal{C}|\ log|\mathcal{C}| + |\mathbb{I}_{\mathcal{T}}|\ |\mathcal{C}|)$.

7 Query Processing & Index Adaptation

This section describes the query processing methods of our approach. Figure 5 outlines the workflow, whose steps are described in the following example.[9] More details on the query evaluation over the index are given in Section 7.1 and on index adaptation in Section 7.2.

Example 9. [***Query Processing & Index Adaptation***] As input we have the initialized index, an exploratory query and a raw file. Considering the objects in Figure 1, we assume an exploratory query Q with the following clauses (left upper corner in Fig. 5) : (1) *Selection clause*: S with S.I_{Long}=[15, 26] & S.I_{Lat}=[5, 17]; (2) *Filter clause*: F = $\{Brand =$ Huawei, $Net =$ 5G$\}$; and (3) *Group-by clause*: G = $\{Provider\}$; and (4) *Analysis clause*: L = $\{count(*)\}$.

Further, we assume that the index is initialized and every tile has a tree with the attributes *Provider* and *Brand*. Additionally, the tree leaves contain aggregate metadata for the *Signal* attribute. The query processing and index adaptation are depicted in Figure 5.

❶ To evaluate query Q we first *find the leaf tiles that spatially overlap* (i.e., partially or fully-contained) with its Selection clause, i.e., t_1, t_2, t_3, t_4. ❷ Next, we *check if the overlapping tiles need to be split*, in such case, the tiles are split into smaller subtiles. The tile splitting may be performed based on different methods, such as: equally or arbitrary-sized splitting. In each splitting step, the process considers criteria related to I/O cost in order to decide whether to perform a split or not (more details at Sect. 7.2). In our example, we assume that t_2 is split into four equal disjoint subtiles: t_{2_a}, t_{2_b}, t_{2_c}, t_{2_d}. ❸ Then, the objects are reassigned to the new subtiles and their *trees are generated*; here, the trees $t_{2_c.h}$ and $t_{2_b.h}$ of the new subtiles t_{2_b} and t_{2_c}.

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

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

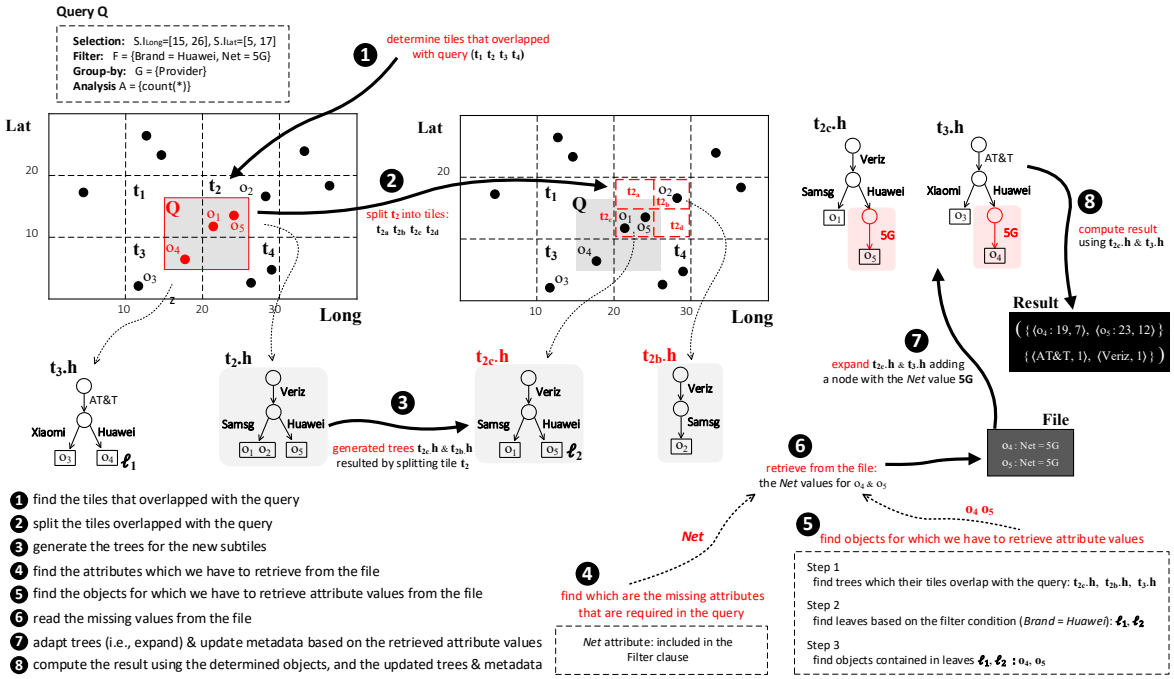


Fig. 5: Query Processing & Index Adaptation Example

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

❼ Based on the values retrieved from the file, *trees are adapted/expanded* (e.g., create new nodes/edges, reorganize leaf objects) in order to include the new attribute and *update the metadata*. Here, using the retrieved *Net* values of o_4 and o_5, the trees $t_{2_c}.h$ and $t_3.h$ are *expanded* to include the new categorical attribute *NET* (see *expand* operation, Sect. 3.1).

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

7.1 Query Processing

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

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

Next, based on the adaptation strategy AS, the adaptTileAndTree procedure (*line* 4), performs the tile splitting and reorganizes the trees (constructing new or modifying existing) that are included in the tiles $\mathcal{T}_a$ created by the splitting process (more details in Sect. 7.2).

Then, considering any conditions over categorical attributes that are defined in the Filter clause, getLeavesBasedOnFilter retrieves the leaf nodes $\mathcal{L}$ of the $\mathcal{T}_a$ trees (*line* 5).

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

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

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

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

1 $\mathcal{L} \leftarrow \varnothing$
2 $\mathcal{T}_\mathsf{S} \leftarrow$ getLeafTilesOverlappedWithQuery ($\mathbb{I}_\mathcal{T}$, S)
3 **foreach** $t_s \in \mathcal{T}_\mathsf{S}$ **do**
4 $\quad \mathcal{T}_a \leftarrow$ AS.adaptTileAndTree (t_s, Q) //see Sect. 7.2
5 $\quad \forall t_a \in \mathcal{T}_a : \ \mathcal{L} \leftarrow \mathcal{L} \bigcup$ getLeavesBasedOnFilter ($t_a.h$, F) //Procedure 2 (Sect. 3.1)
6 $\mathcal{W}(\langle l, \mathcal{V}\rangle) \leftarrow$ getLeavesRequiringFileAccess ($\mathcal{L}$, Q) //set of tuples, where $\mathcal{V}$ are the attributes of a leaf l whose their values need to be retrieved from the file
7 **if** $\mathcal{W} \neq \varnothing$ **then** //values are missing — read from file
8 $\quad$ *read from file* the values of attributes $\mathcal{V}$ for the objects of leaf l, $\forall\langle l, \mathcal{V}\rangle \in \mathcal{W}$
9 $\quad$ expandTree (l, $\mathcal{V}$) $\forall\langle l, \mathcal{V}\rangle \in \mathcal{W}$ //update tree based on retrieved attributes; i.e., expand tree's leaf l with its missing attributes $\mathcal{V}$ (Sect. 3.1)
10 $\quad$ updateLeafMetadata (l) $\forall l \in \mathcal{W}$
11 $\mathcal{R} \leftarrow$ *evaluate* Q using the objects and the metadata of leaves $\mathcal{L}$
12 **return** $\mathcal{R}$

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

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

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

Adapt Trees and Update Metadata based on the Attributes Read from File. Next, based on the attributes for which values are read from the file, the trees (of fully-contained tiles) are adapted/enriched to include the retrieved attributes. Particularly, the expandTree procedure (Sect. 3.1) adapt/enriches the trees by including the retrieved attributes and reorganizes the objects (*line* 9). Then, the function updateLeafMetadata computes and updates the metadata using the values retrieved from the file (*line* 10).

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

7.2 Incremental Index Adaptation

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

Procedure adaptTileAndTree (*line* 4) is responsible for the incremental adaptation. It takes as input a tile t and a query Q and returns a set of subtiles $\mathcal{T}_a$ if the tile t needs to be split.

To split, or not to split? During query processing, we examine each tile that overlaps with the query if it needs to be split. To determine if a tile t requires (further) splitting, we define a model that estimates the *expected splitting gain* in terms of I/O cost, for evaluating a (future) query Q, in case of splitting t. If the expected splitting gain for a tile, exceeds a given splitting threshold, a split is performed. A further analysis of the splitting model is presented in [12].

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

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

In our implementation for VETI, we employ a quad-tree like splitting approach in which a tile is split into 4 equal subtiles. However, more sophisticated methods can be used to split a tile, e.g. query based splitting methods [12].

Reorganize Trees in Splitted Tiles. As discussed in Section 3.1, the order of the attributes in a tree affects its size (number of nodes/edges). Hence, during splitting, the attributes of the trees that are generated in the new subtiles, are sorted so that the attributes with smaller domain sizes are placed closer to the root. For this, we consider the distinct values of the categorical attributes within the bounds of the parent tile t. Then, we reorganize the objects of t into the trees of the children $\mathcal{T}_a$.

8 Experimental Analysis

The primary objective of our evaluation is to assess the performance of our approach in terms of initialization and exploration time, and number of I/Os. We evaluate different VETI variations and several competitors over 2 real and 2 synthetic datasets; by conducting experiments w.r.t. data characteristics and available resources, i.e., the number and the domain size of the categorical attributes, as well as the initial memory allocated for the index. In what follows, we outline the key findings of our study.

Results Highlights. (1) *Performance Overview*: In most queries, VETI exhibits *response time less than* 0.04sec, over large raw files (e.g., 45GB). Regarding the best of the examined systems, in most queries VETI *is up to 100× faster* and performs up to *2 orders* of magnitude fewer I/O operations.

(2) *Data Characteristics*: All VETI variations report (sub-)linear performance w.r.t. the number of objects and categorical attributes, as well as the domain size.

(3) *VETI Variations*: Regarding the VETI variations, both VETI-BINN and VETI-GRD outperform the naive VETI-RND. VETI-BINN is more than 1.5× faster and requires about half the I/O operations compared to VETI-GRD. VETI-BINN performs even better when the user moves further away from the initial query and/or when the initialization budget is small.

(4) *Initialization Phase*: In the initialization phase, VETI-BINN is on average 8× faster than MySQL, 1.2× faster than PRAW, and slightly slower than VALINOR.

8.1 Experimental Setup

Datasets & Queries. In our experiments we have used two *real datasets*, the *NYC Yellow Taxi Trip Records* (TAXI) and *a telecommunication network quality dataset* (NET); and two synthetic ones (SYNTH10 / 50).[10]

Queries Template. Each query contains: (1) a *Select* clause defined over the axis-attributes; (2) a *Group-by* clause on a categorical attribute; (3) a *Filter* clause that contains either 1 or 2 equality conditions, specified over randomly selected categorical attributes and values from their corresponding domains; and (4) an *Analysis* clause that computes 5 aggregate functions over a numeric attribute, i.e., min, max, std, variance, and mean.

TAXI Real Dataset. The TAXI dataset is a csv file, containing information regarding taxi rides in NYC.[11] Each record corresponds to a trip, described by 18 *attributes*, e.g., pick-up and drop-off dates and locations, trip distances, fares. From these attributes, 5 are categorical: Payment Type, Passenger Count, Rate Type, Provider Code and Store & Forward Flag. We selected a subset of this dataset for 2014 trips with 165M objects and 26 GB raw data file size.

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

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

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

SYNTH10 / 50 Synthetic Datasets & Queries. Regarding the *synthetic datasets* (SYNTH10 / 50), we have generated two csv files of 100M *data objects* (in the default setting), having 10 and 50 *attributes* (11 and 51 GB, respectively). The datasets contain *numeric attributes* in the range [0, 1000], as well as *categorical attributes*, where the values of the numeric and categorical attributes follow a uniform distribution. In our experiments, we vary the *number of objects* from 1M to 500M, objects with 100M being the default value, where the size of the dataset having 500M objects is 52GB. Regarding queries, as in [12], the *Select* clause is defined over two numeric attributes that specify a window size containing approximately 100K objects.

Exploration Scenario. In our evaluation, we consider a typical exploration scenario such as the one described in the introduction (Sect. 1), in which the user explores different areas, and also filters, and performs Group-by operations. We have generated sequences of 100 overlapping queries, with each window query shifted (i.e., pan operation) in relation to its previous one by 10% towards a random direction.

[10] The data generator for the SYNTH10/50 along with the queries for the TAXI and SYNTH10/50 datasets, are available at: github.com/VisualFacts/RawVis

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

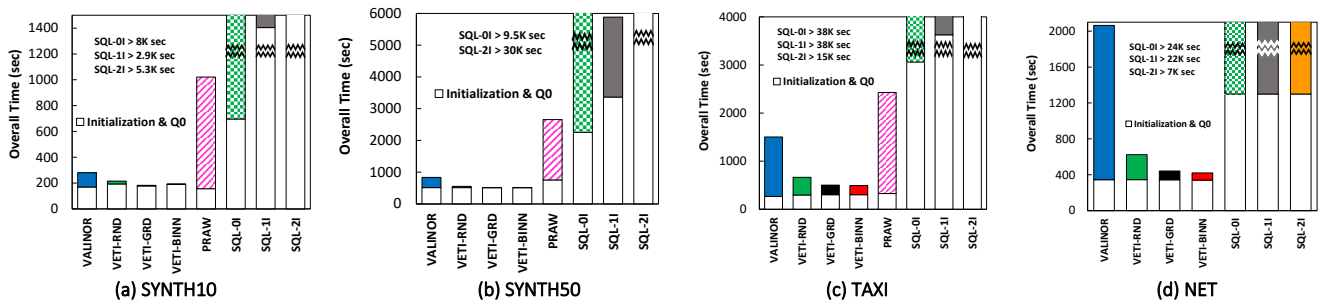


Fig. 6: Overall Time (Broken down to Initialization & $Q_1 \sim Q_{99}$ Evaluation Time)

This scenario attempts to formulate a common user behavior in 2D visual exploration, where the user explores nearby regions using pan and zoom operations. [53,29,47, 8,51,16], like the "region-of-interest" or "following-a-path" scenarios commonly used in map-based visual exploration.

Additionally, to formulate the "repetitive calculation of statistics" that commonly appears in exploration scenarios (Sect. 5.1) [50], we included the attributes of the initial query in the generated sequence of queries four times more frequently than the other dataset attributes.

VETI Parameters. Regarding VETI's tile structure, we adopt the setting used in [12], where the tile structure is initialized with 100×100 equal-width tiles, while an extra 20% of the number $|\mathcal{T}_0|$ of initial tiles was also distributed around the first query Q_0 using the Query-driven initialization method [12]. Also, the numeric threshold for the adaptation of VETI was set to 200 objects. More details about these parameters can be found at [12].

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

VETI Variations. We evaluate two versions of VETI, named VETI-GRD and VETI-BINN, based on the GRD and BINN algorithms (Sect.6). Moreover, to further demonstrate the effectiveness of these algorithms, we consider a naive assignment approach, titled VETI-RND, which follows a random tile-tree assignment strategy. It first sorts the tiles based on the tile probability ρ_t, but then assigns a randomly selected tree from the entire powerset $HP_{\mathcal{C}}$ to each tile, until the budget constraint $\mathcal{B}$ is satisfied.

Competitors. We compare our method with: (1) VALINOR [12] which contains only the tile-based indexing structure without the CET index; (2) A traditional DBMS (MySQL 8.0.22), where data is loaded and indexed in advance before users start executing 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. 9); however, due to low performance [7], we do not consider it as a competitor in our evaluation. (3) PostgresRaw (PRAW)[12], built on top of Postgres 9.0.0 [7], which is a generic platform for in-situ querying over raw data (more details at Sect. 9). Note that, due to parsing/processing problems on the NET dataset with the PRAW, we did not manage to load and report experiments on this combination.

Table 3: Basic Parameters

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

Metrics. In our experiments, we measure the: (1) *Evaluation Time* of a query; (2) *Initialization Time*, which corresponds to the time required to initialize the index and return the results of the first query Q_0, i.e., from-raw data-to-1st result time. Regarding the initialization phase of the examined systems we have: (a) before evaluating Q_0, MySQL needs to parse the raw file, load, and index (except SQL-0I) the data; (b) during evaluating Q_0, PRAW needs to parse the raw file and construct the positional map; (c) during evaluating Q_0, VALINOR parses the raw file, generates the tile index structure, and populates it with the object entries; and (d) during evaluating Q_0, beyond the actions performed by VALINOR, VETI also parses the categorical attributes and constructs the tree indexes over the tiles. (3) *Overall Execution Time* of an exploration scenario, that includes: initialization time and query evaluation time for all the queries included in the exploration scenario, i.e., workload; (4) *I/O Operations* performed during query evaluation (for I/O definition see Sect. 9); and (5) *Index Utility*.

Implementation. VETI is implemented on JVM 1.8 as part of the RawVis *open source data visualization system* [35], and the code is available under GNU/GPL.[2] The experiments were conducted on an 3.60GHz Intel Core i7-3820

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

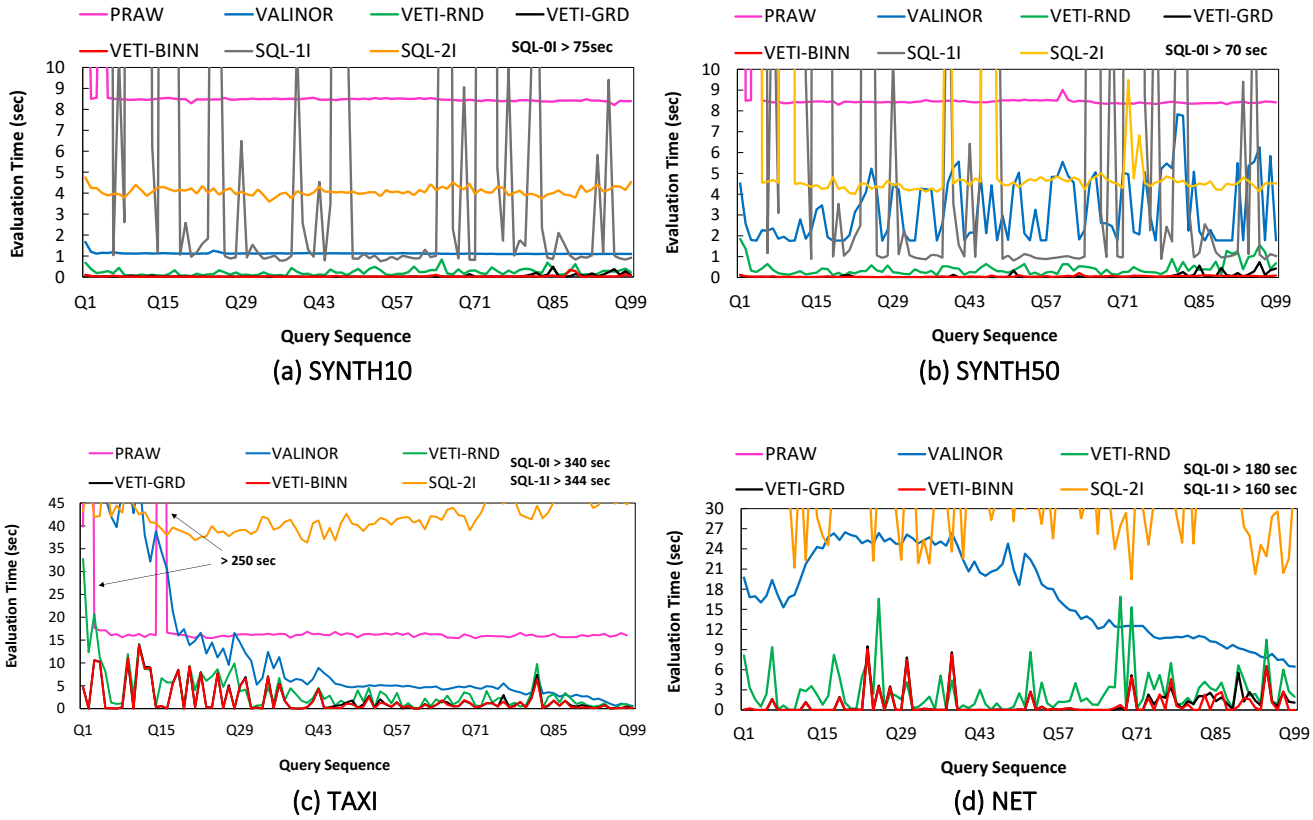


Fig. 7: Evaluation Time per Query (sec)

with 64GB of RAM. We applied memory constraints (32GB max Java heap size) in order to measure the performance of our approach and our competitors. However, PRAW required more than 32GB of memory for the synthetic datasets and more than 50GB for the TAXI dataset.

Table 3 summarizes the parameters that we vary in the experiments and the range of values examined; default values are presented in bold.

8.2 Performance

Initialization Phase: From-Raw Data-to-1st Result Time. Figure 6 presents the overall execution time which is split between the initialization time and the time for evaluating all the queries $Q_1 \sim Q_{99}$. Recall that, the initialization time includes the time for parsing, loading the data (in the case of MySQL), constructing the index and answering the first query Q_0. In Figure 6 we can observe that the MySQL settings we examined exhibit the worst performance for evaluating the first query, since MySQL needs to parse all attributes of the raw file and load the data in the disk. Also, for the SQL-1I and SQL-2I cases, the corresponding indexes must be built, which explains the increased initialization time in relation to SQL-0I where no index is generated. Both VALINOR and the VETI variations exhibit better initialization performance compared to PRAW for the SYNTH50 and TAXI datasets, while for the SYNTH10 dataset, VETI requires a slightly higher initialization time. As it is expected, VETI variations are slightly slower during the initialization compared to VALINOR, since VETI needs to determine the tile-tree assignments, parse the categorical attributes, and create the tree structures. All VETI variations, however, exhibit similar initialization time, since the tile-tree assignment time is negligible compared to the time for parsing the file.

Evaluation Time per Query. Figure 7 presents the evaluation time for each individual query. Compared to the other methods, all VETI variations exhibit significantly lower evaluation time in almost all queries and datasets. In most queries, VETI reports *evaluation time less than* 0.04sec. On the other hand, the best competitors, PRAW and VALINOR require for most queries more than 8 and 4sec, respectively. Overall, VETI *is more than 200× and 100× faster compared to* PRAWand VALINOR, respectively.

Regarding SQL, SQL-0I performs worse than the 3 SQL settings we examined, and requires approximately the same time for each query. This is expected as SQL-0I has no index. From the other 2 settings, SQL-1I is for most queries faster than SQL-2I for the two synthetic datasets, and slower for TAXI and NET.

Regarding PRAW, we observe that it exhibits a stable performance (after the first queries), which is however worse than both VALINOR and VETI in all datasets. The positional map used in PRAW, attempts to reduce the parsing and tokenizing costs of future queries, by maintaining the position of specific attributes for every object in the raw

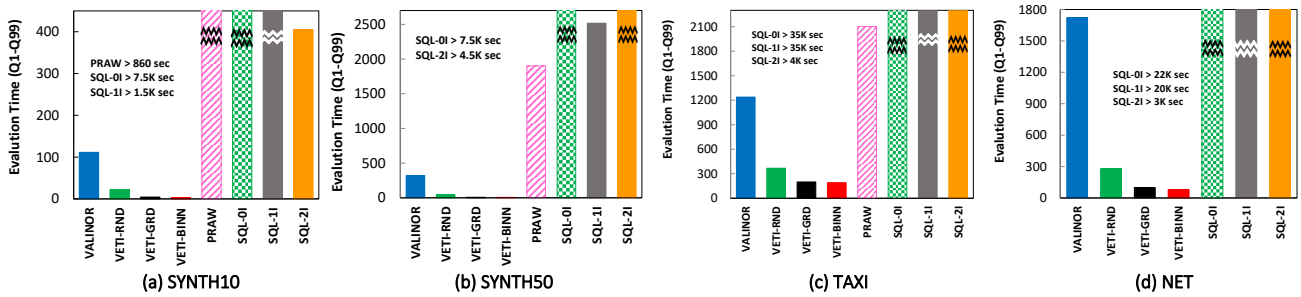


Fig. 8: Evaluation Time for $Q_1 \sim Q_{99}$ (sec)

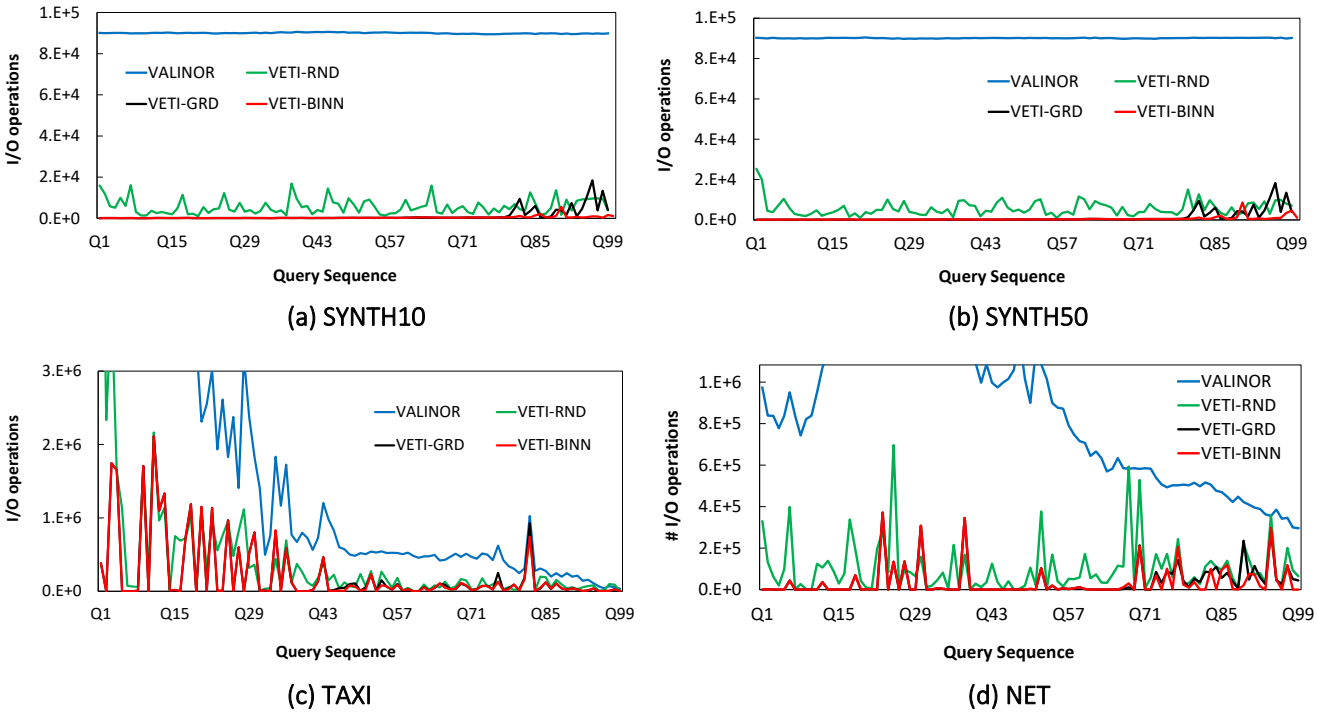


Fig. 9: Number of I/Os per Query

file. However, PRAW still needs to examine all objects in the dataset in order to select the ones contained in a 2D window query. Also, in contrast to VETI, PRAW does not keep any metadata in order to efficiently compute the aggregate queries. Some of the early queries (approximately until Q_{15}) PRAW exhibits noticeably higher time than the rest, and comparable to the time required to answer Q_0. This is due to the filter conditions of the queries. When a query refers to an attribute that was not included in Q_0, PRAW needs to populate the positional map with it. In subsequent queries, which refer to indexed attributes, PRAW exhibits a relatively constant evaluation time.

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

Evaluation Time for all $Q_1 \sim Q_{99}$ Queries. Figure 8 presents the evaluation time for the $Q_1 \sim Q_{99}$ queries. The behavior of the methods is similarly between the datasets, the variations of VETI significantly outperform the competitors. The best competitor, VALINOR needs about 30, 60, 7 and 20$\times$ more time for SYNTH10, SYNTH50, TAXI and NET, respectively, to evaluate all queries. Also, PRAW is 270, 380 and 11$\times$ slower for SYNTH10/50 and TAXI, respectively.

I/O Operations. The evaluation time for VETI and VALINOR is mainly determined by the number of rows that need

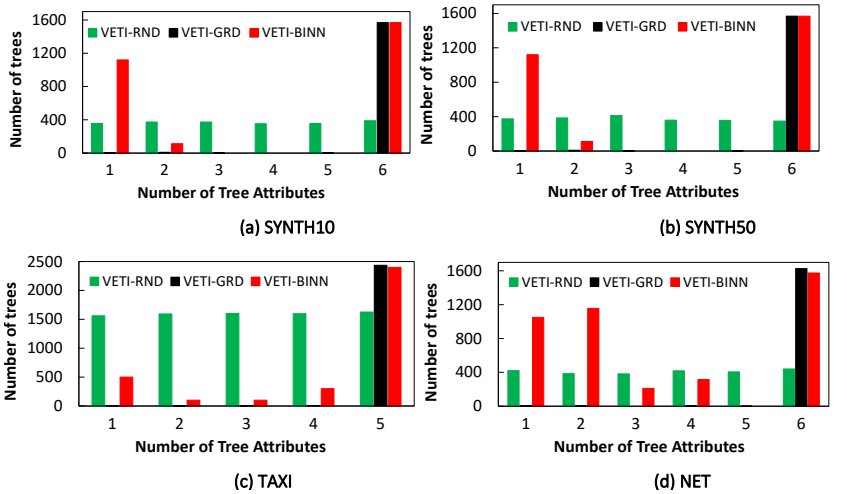


Fig. 10: Number of Generated Trees v.s. Number of Tree Attributes

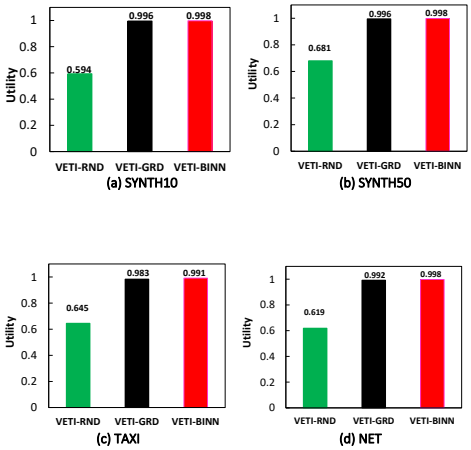


Fig. 11: VETI Utility Score

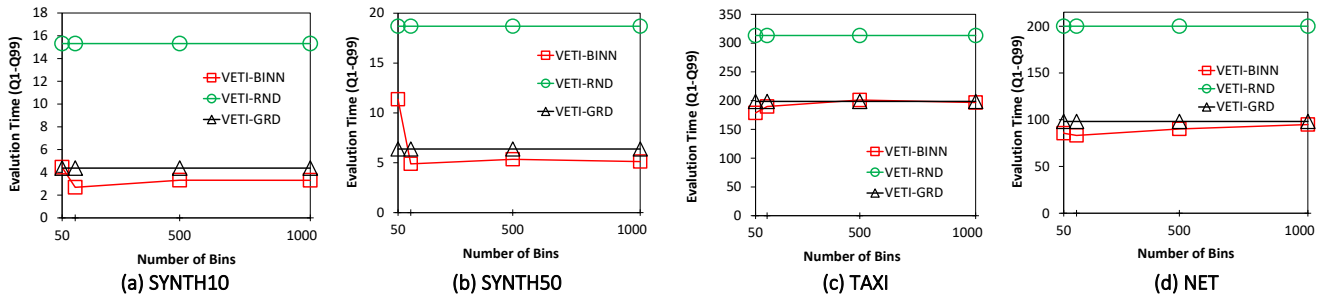


Fig. 12: VETI-BINN: Evaluation Time for $Q_1 \sim Q_{99}$ (sec) vs. Number of Bins

to be retrieved from the raw file, i.e., number of I/O operations. This can be observed in Figure 9, where the number of I/O operations per query exhibits approximately the same behavior with that of the evaluation time (Fig. 7). Note that we do not present the I/Os for PRAW and SQL, since they follow different workflows/methods for accessing the file, compared to our work. Compared to VALINOR, the VETI variations perform up to 2 orders of magnitude less I/Os, during. This occurs since VALINOR has to access the raw file for every object contained in the 2D window query in order to retrieve the categorical attribute values required by the query.

8.3 VETI Variations

Performance & Assignments. Here, we compare the performance of the three VETI initialization variations. Overall, considering the performance of VETI variations (Fig. 7), both VETI-GRD and VETI-BINN lead to faster query responses than the naive VETI-RND, in almost all cases. Also, VETI-BINN significantly outperforms the other two in the number of I/Os (Fig. 9). Considering the time required for all queries (Fig. 8), VETI-RND is on average 3× slower than VETI-GRD and VETI-BINN. Comparing VETI-GRD and VETI-BINN, VETI-BINN is more than 1.5× faster than VETI-GRD (in some cases more than 100× faster) (Fig. 8), and performs more than 3× less I/Os.

The difference in performance is the result of the different assignment policies (see Sect.6.3 for the policies used in VETI-BINN). Figure 10 depicts the number of trees generated during the initialization w.r.t. the number of attributes they have. We can observe, that VETI-RND follows a uniform distribution w.r.t. the number of tree attributes. In VETI-GRD the budget is mostly allocated at constructing trees that contain all of the categorical attributes. In contrast, VETI-BINN creates a more balanced distribution of the trees' number of attributes. As a result, VETI-GRD assigns "taller" trees to a smaller number of tiles. So, due to location-based assignments process we follow, the tiles located farther away from Q_0 tend to not contains trees. This is why, compared to VETI-GRD, VETI-BINN tends to perform even better when the user moves away from the initial starting point. This is demonstrated, in the query performance where, in most cases, after query Q_{75}, VETI-BINN is 10× faster than VETI-GRD in (Fig. 7a, b, d); also, in some queries is up to 400× faster (Fig. 7d).

The impact of the different assignment strategies is also shown in the utility score (Fig. 11). Due to randomized tree assignments, VETI-RND results in a lower utility score, whereas VETI-BINN exhibits larger utility compared to VETI-GRD.

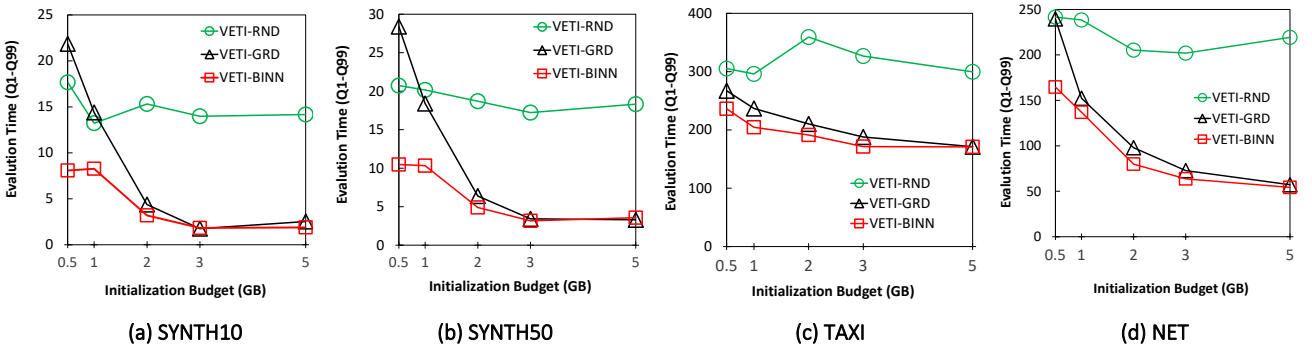


Fig. 13: Evaluation Time for $Q_1 \sim Q_{99}$ (sec) vs. Initialization Memory Budget

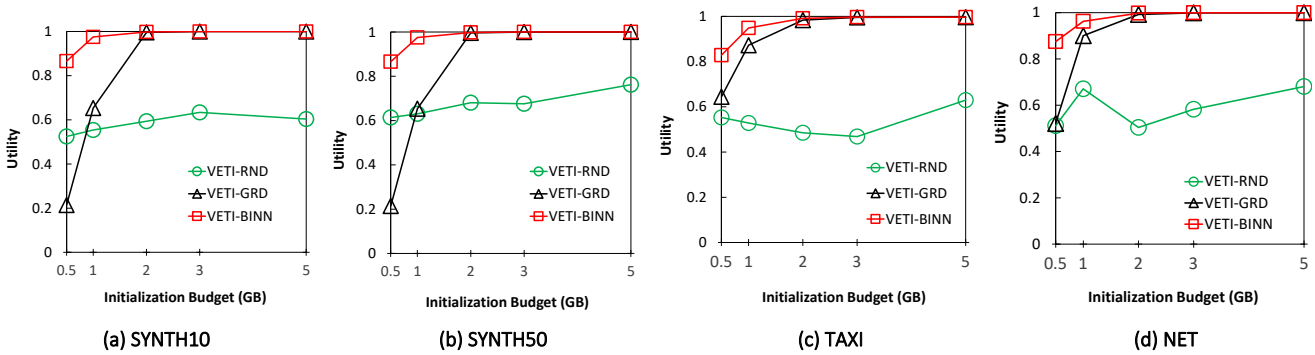


Fig. 14: VETI Utility vs. Initialization Memory Budget

VETI-BINN: Varying the Number of Bins. In this experiment we study the performance of VETI-BINN w.r.t. the number of bins. Figure 12 presents the evaluation time for $Q_1 \sim Q_{99}$, varying the number of bins from 50 to 1000. Note that, in the plots we include VETI-RND and VETI-GRD for the sake of comparison, even though they do not depend on the number of bins.

As we can observe, the performance of VETI-BINN is not highly affected by the number of bins, except for small number of bins, i.e., between 50 and 100. Based on our adopted assignments policies for the VETI-BINN (Sect. 6.3), the following holds. For small number of bins, the assignment is more coarse-grained, i.e., shorter trees are assigned to the majority of the tiles. Increasing the number of bins results in more fine-grained assignments of trees to (bins of) tiles. However, trees that are assigned to bins near Q_0 will be taller, whereas the trees assigned to the remaining bins will be short. This explains why, in general, as the number of bins increase, the performance of VETI-BINN approaches that of VETI-GRD. Recall that VETI-GRD assigns mostly tall trees (Fig. 10).

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

Varying the Initialization Memory Budget. In this experiment, we evaluate the performance of VETI while varying the initialization budget from 0.5 to 5GB. Recall that this memory budget includes only the memory allocated by the tile and tree structures, and does not include the memory required to store the object entries.

The evaluation time needed to evaluate all the queries is shown in Figure 13. The evaluation time decreases as the available memory budget increases. This is the result of the larger number and more detailed tree structures that are constructed with more budget, which leads to faster query evaluation. This is observed in both VETI-GRD and VETI-BINN. Regarding VETI-RND, its performance does not always improve when increasing the budget, as it allocates the budget in random tile-tree assignments.

Compared to VETI-GRD, VETI-BINN's performance is less dependent on the available budget. VETI-GRD performs much worse for low values of memory budget, as it mostly assigns trees with all the categorical attributes included which quickly depletes the budget on very few tiles. On the other hand, as the budget increases, we can observe that the performance of VETI-GRD is comparable with that of VETI-BINN.

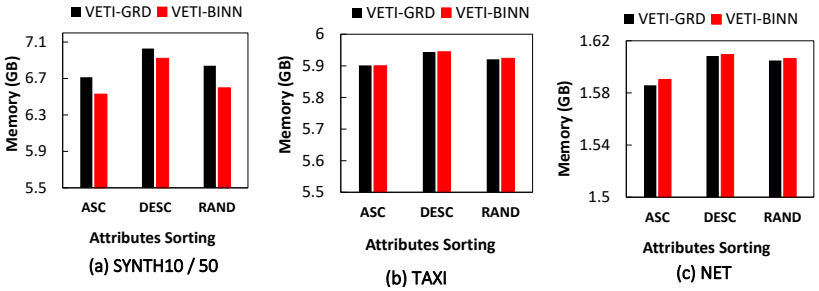
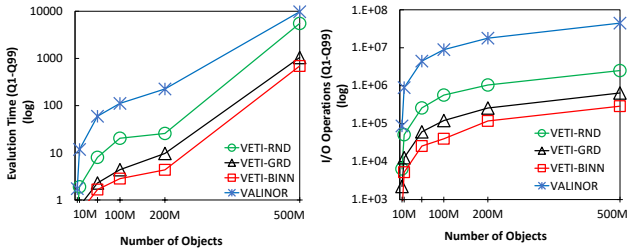
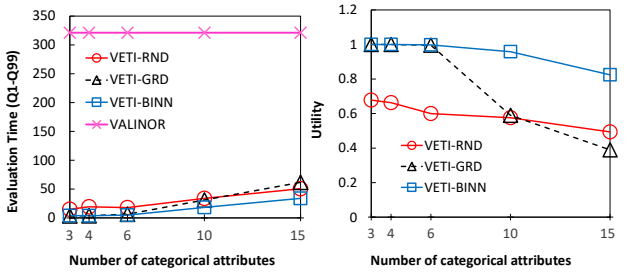


Fig. 15: Memory Size vs. Sorting Attributes based on Domain Size



(a) Evaluation Time (log) (b) I/Os (log)

Fig. 16: Varying the Number of Objects [SYNTH10]



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

Fig. 17: Varying the Number of Cat. Attributes [SYNTH50]

In more details, on small amounts of budget, VETI-BINN evaluates all the queries of the exploration scenario in half the time compared to VETI-GRD (Fig. 13). Moreover, at query level, in several queries, compared to VETI-GRD, VETI-BINN is from 100 to 3000× faster for the NET dataset; and more than 50× for SYNTH10/50 (the plot is omitted for brevity).

Figure 14 presents the utility score. The results closely match the evaluation time presented above. Specifically, the total index utility increases with higher budget for both VETI-GRD and VETI-BINN. Also, the utility of VETI-GRD is much lower than that of VETI-BINN for smaller amounts of budget, but their values converge as the budget increases.

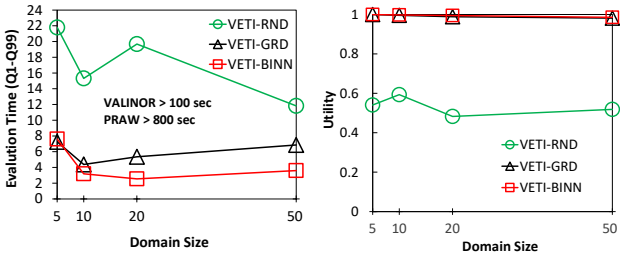
VETI Memory Size vs. Sorting Tree Attributes. In this experiment, we examine how the sorting of tree attributes w.r.t. their domain size affects the memory size. Recall that, the order of the attributes in the tree levels w.r.t. their domain size, determines the number of tree nodes, and as a result the tree memory footprint (Sect. 3).

For this experiment we used the real datasets, as well as a version of the SYNTH10 where each of its 6 categorical attributes had a different domain size, varying from 2 to 100. We measured the VETI memory size after initialization while sorting the attributes based on their domain sizes in ascending (ASC), descending (DESC) and random (RAND) order. This memory corresponds to the total size of the index, with its tiles, trees and objects entries. As it can be seen in Figure 15, ASC ordering corresponds to the best case w.r.t. memory consumption, while DESC to the worst. For example, in SYNTH10 which had the largest variation in the domain size of its categorical attributes, VETI-BINN requires about 6.5GB, 6.6GB and 7GB for ASC, RAND and DESC respectively. Similar results are reported for VETI-GRD.

8.4 Effect of the Data Characteristics

Varying the Number of Objects. In this experiment, we evaluate the impact of the number of objects on the performance of VETI. For this, we vary the number of objects of SYNTH10 from 5 to 500M, and the evaluation time for $Q_1 \sim Q_{99}$ are presented in Figure 16(a). As the total number of objects in the file increase, the evaluation time increases (sub-)linearly for all variations of VETI as well as for VALINOR. This is reasonable considering that the index becomes more dense, the queries relatively select more objects and the number of required I/O operations increase (Fig. 16(b)); also, the cost of an I/O operation becomes "relatively" larger when the file size increase. Regarding VETI-RND, its performance is affected to a much greater extent compared to VETI-GRD and VETI-BINN, as its randomized tree assignment lead to a much higher I/O cost.

Varying the Number of Categorical Attributes. In this experiment, we vary the number of categorical attributes (Fig. 17). Here, we used the SYNTH50 dataset in order to be able to select up to 15 categorical attributes. For brevity the SQL and PRAW methods are omitted, as they exhibit much higher evaluation time. Also, we could not evaluate PRAW for 15 categorical attributes, due to increased memory requirements. Note that VALINOR's performance is not affected by the number of categorical attributes, since it does not consider them in its index structure.



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

Fig. 18: Varying the Domain Size of Categorical Attributes [SYNTH10]

As we can observe, query evaluation time increases for all methods, along with the number of attributes (Fig. 17(a)). VETI-BINN outperforms both VETI-RND and VETI-GRD. Regarding VETI-GRD, we can observe that it outperforms VETI-RND in every case except for the case of 15 categorical attributes. This is due to the fact that VETI-GRD allocates most of the budget for creating trees with all the categorical attributes. As a result, with a higher number of categorical attributes, VETI-GRD assigns trees to very few tiles, which explains its performance deterioration for 15 attributes. This is also depicted in Figure 17(b) which presents the utility score. As we can observer, in all VETI variations the utility score decreases as the number of categorical attributes indexed increase. This decrease is even more notable in the case of VETI-GRD, which after 10 attributes gets worse than both VETI-BINN and VETI-RND.

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

The evaluation time needed to execute the $Q_1 \sim Q_{99}$ is shown in Figure 18(a). Note that the plot shows only the VETI variations, since VALINOR does not depend on the domain, and others exhibit much higher evaluation time. We can observe that the evaluation time of VETI-GRD (resp. VETI-BINN) decreases from domain size 5 to 10 (resp. 20), and then increases.

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

On the other hand, given the same number of attributes, a larger domain size results in trees with larger size in memory (Sect. 3.1). This explains the increase in evaluation time after domain size 10 for VETI-GRD and 20 for VETI-BINN, as the larger tree sizes result in fewer tiles getting assigned with trees.

9 Related Work

In-situ Raw Data Processing. Data loading and indexing usually take a large part of the overall time-to-analysis for both traditional RDBMs and Big Data systems [24]. In-situ query processing aims at avoiding data loading in a DBMS by accessing and operating directly over raw data files. NoDB [7] was a first approach for a no-dbms querying engine over raw data, and PostgresRAW is one of the first systems 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 VETI, 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, VETI is better optimized for aggregate queries, since it can reduce raw file accesses by reusing already calculated statistics on a tile level. Finally, VETI offers memory-driven index initialization, by adjusting its initial parameters w.r.t. available resources.

DiNoDB [48] is a distributed version of PostgresRAW. In the same direction, PGR [31] extends the positional maps in order to both index and query files in formats other than CSV. In the same context, Proteus [30] supports various data models and formats. Recently, Slalom [41,42] 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 [26] and SDS/Q [14] rely on DBMS technologies to perform analysis over scientific array-based file formats. Further, SCANRAW [15] considers parallel techniques to speed up CPU intensive processing tasks associated with raw data accesses.

[12,11] define VALINOR, a tile-based index in the context of in-situ visual exploration, supporting 2D visual operations over numeric attributes. Compared to VETI, VALINOR does not support operations and indexing over categorical attributes and does not consider memory-driven index initialization. As a result, it cannot exploit well-known visualization techniques, such as bar charts, heatmaps, pies and parallel coordinates. Particularly, VETI extends a tile-based structure with trees that enrich tiles with information about categorical attributes. Also, the index structures and the methods proposed here are defined based on available memory resources.

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

cessing of a specific query class, that enables user operations and analytics in 2D visual exploration scenarios, e.g., pan, zoom, aggregate analytics. 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-Oriented Indexes. In the context of visual exploration, several indexes have been introduced. VisTrees [17] and HETree [13] are tree-based main-memory indexes that address visual exploration use cases, i.e., they offer exploration-oriented features such as incremental index construction and adaptation. Compared to our work, both indexes focus on one-dimensional visualization techniques (e.g., histograms), do not support categorical attributes and group-by analytics, and do not consider disk storage, i.e., data stored in-memory.

Nanocubes [33], Hashedcubes [32], SmartCube [34], Gaussian Cubes [49], and TopKubes [37] are main-memory data structures defined over spatial, categorical and temporal data. The aforementioned works are based on main-memory variations of a data cube in order to reduce the time needed to generate visualizations. Nanocubes [33] attempts to reduce the memory of the data cube by sharing nodes in a single tree structure. Hashedcubes [32] follows a different approach where, instead of materializing all possible aggregations, it uses a partial ordering of the dimensions and the notion of pivot arrays to calculate on-the-fly the aggregations missing. Smartcube [34] is a variation of Nanocubes, where instead of pre-computing all cuboids from the start, it chooses some important ones based on the user queries, in order to reduce memory usage. Also, it may adaptively change stored cuboids when querying patterns change. In comparison with our work, the indexes in the aforementioned works are generated during a preprocessing phase, and thus cannot be used in in-situ scenarios, e.g., they do not address problems related to reducing the initialization time. Moreover, a major difference compared to our approach, is that these works assume that all the aggregations are materialized and stored in main memory, which in most cases require prohibitive amounts of memory. In contrast, our approach considers limited memory resources.

Further, graphVizdb [10,9] 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.

In a different context, tile-based structures are used in visual exploration scenarios. Semantic Windows [29] considers the problem of finding rectangular regions (i.e., tiles) with specific aggregate properties in exploration scenarios. ForeCache [8] 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. Our work considers different problems compared to the aforementioned approaches.

We should note that there are a large number of spatial indexes such as the R-tree, kd-tree, quadtree [20] which could be used in the context of data exploration. However, most of these structures consider several criteria (e.g., tree balance, fill guarantees) in order to improve query processing, which results in a significant amount of time required for their construction [12]. As a result, they are not suitable for the in-situ setting, which requires small initialization overhead.

Adaptive Indexing. Similarly to VETI, the basic idea of approaches like database cracking and adaptive indexing is to incrementally adapt the indexes and/or refine the physical order of data, during query processing, following the characteristics of the workload [27,22,52,25,40,22,23,39,43].

However, these works are not designed for the in-situ scenario. In most cases the data has to be previously loaded / indexed in the system/memory, i.e., a preprocessing phase is considered. Additionally, the aforementioned works refine the (physical) order of data, performing highly expensive data duplication and allocate large amount of memory resources. Nevertheless, in the *in-situ scenarios* the analysis is performed directly over immutable raw data files considering limited resources.

Furthermore, most of the existing cracking and adaptive indexing methods have been developed in the context of column-stores or MapReduce systems [46]. On the other hand, VETI has been developed to handle raw data stored in text files with commodity hardware.

Progressive Visualization. Recently, many systems adopt the *progressive paradigm* attempting to reduce the response time [18,45,6,19]. 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 improve with time. Therefore, instead of waiting for an unbounded amount of time, users can see the results unfolding progressively. Compared to these works, we do not consider progressive and approximate computations; rather exact answers are presented to the users as soon as they are computed. However, we find most of these approaches quite relevant and could be exploited in our future work in the computation of complex statistics over the index.

10 Conclusions

In this paper, we presented our work that enables efficient in-situ visual exploration and analysis of data visualized in a 2D plane (e.g., map, scatter plot), supporting visual user interactions, categorical based visual analytics, and computations of statistics. We have presented an indexing scheme and adaptive processing methods that allow users to combine visual exploration of data from a raw file on a 2D visual representation with sophisticated analysis over its numeric, spatial, and categorical attributes. This scheme is constructed on-the-fly given the first user query; during the exploration the index is progressively adapted based on user interactions and analytical tasks.

To handle the large memory requirements, we formulated the index initialization as an optimization problem,

which determines the initial index characteristics w.r.t. available memory resources. We have shown that the problem is NP-hard, and we have provided two approximate algorithms for its solution. We have further presented efficient query evaluation methods that achieve fast user response by reusing available metadata stored in the index, avoiding I/O operations. The proposed methods have been implemented as part of RawVis open source data visualization system, and the source code is available under GNU/GPL.

Finally, we have conducted a thorough experimental evaluation with real and synthetic datasets for assessing the performance of the proposed techniques. The results demonstrate that the proposed methods significantly outperform existing solutions in terms of I/O operations and query response time.

Acknowledgment. This work has been funded by the project VisualFacts (#1614 - 1st Call of the Hellenic Foundation for Research and Innovation Research Projects for the support of post-doctoral researchers).

References

1. MySQL: The CSV Storage Engine. https://dev.mysql.com/doc/refman/8.0/en/csv-storage-engine.html
2. Oracle: External Table Enhancements in Oracle Database 12c Release 1. https://oracle-base.com/articles/12c/external-table-enhancements-12cr1
3. PostgreSQL: Foreign Data. https://www.postgresql.org/docs/current/ddl-foreign-data.html
4. SciPy: Scientific Tools for Python. http://www.scipy.org
5. Wolfram : Descriptive Statistics. https://reference.wolfram.com/language/tutorial/DescriptiveStatistics.html
6. Agarwal, S., Mozafari, B., Panda, A., Milner, H., Madden, S., Stoica, I.: Blinkdb: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In: European Conference on Computer Systems (EuroSys) (2013)
7. Alagiannis, I., Borovica, R., Branco, M., Idreos, S., Ailamaki, A.: Nodb: Efficient Query Execution on Raw Data Files. In: SIGMOD (2012)
8. Battle, L., Chang, R., Stonebraker, M.: Dynamic Prefetching of Data Tiles for Interactive Visualization. In: SIGMOD (2016)
9. Bikakis, N., Liagouris, J., Krommyda, M., Papastefanatos, G., Sellis, T.: Towards Scalable Visual Exploration of Very Large Rdf Graphs. In: ESWC (2015)
10. Bikakis, N., Liagouris, J., Krommyda, M., Papastefanatos, G., Sellis, T.: Graphvizdb: A Scalable Platform for Interactive Large Graph Visualization. In: ICDE (2016)
11. Bikakis, N., Maroulis, S., Papastefanatos, G., Vassiliadis, P.: RawVis: Visual Exploration over Raw Data. In: Advances in Databases and Information Systems (ADBIS) (2018)
12. Bikakis, N., Maroulis, S., Papastefanatos, G., Vassiliadis, P.: In-situ Visual Exploration over Big Raw Data. Information Systems **95** (2021)
13. Bikakis, N., Papastefanatos, G., Skourla, M., Sellis, T.: A Hierarchical Aggregation Framework for Efficient Multilevel Visual Exploration and Analysis. Semantic Web Journal (2017)
14. Blanas, S., Wu, K., Byna, S., Dong, B., Shoshani, A.: Parallel Data Analysis Directly on Scientific File Formats. In: SIGMOD (2014)
15. Cheng, Y., Rusu, F.: SCANRAW: a Database Meta-operator for Parallel In-situ Processing and Loading. TODS **40**(3) (2015)
16. Dar, S., Franklin, M.J., THór Jónsson, B., Srivastava, D., Tan, M.: Semantic Data Caching and Replacement. In: VLDB (1996)
17. El-Hindi, M., Zhao, Z., Binnig, C., Kraska, T.: Vistrees: Fast Indexes for Interactive Data Exploration. In: HILDA (2016)
18. Fekete, J., Fisher, D., Nandi, A., Sedlmair, M.: Progressive Data Analysis and Visualization. Dagstuhl Reports **8**(10) (2018)
19. Fisher, D., Popov, I.O., Drucker, S.M., Schraefel, M.C.: Trust Me, I'm Partially Right: Incremental Visualization Lets Analysts Explore Large Datasets Faster. In: CHI (2012)
20. Gaede, V., Günther, O.: Multidimensional Access Methods. ACM Comput. Surv. **30**(2) (1998)
21. Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., Pirahesh, H.: Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals. Data Min. Knowl. Discov. **1**(1) (1997)
22. Holanda, P., Manegold, S.: Progressive mergesort: Merging batches of appends into progressive indexes. In: EDBT (2021)
23. Holanda P., Manegold S., Mühleisen H., Raasveldt M: Progressive Indexes: Indexing for Interactive Data Analysis. PVLDB 2019
24. Idreos, S., Alagiannis, I., Johnson, R., Ailamaki, A.: Here Are My Data Files. Here Are My Queries. Where Are My Results? In: CIDR (2011)
25. Idreos, S., Kersten, M.L., Manegold, S.: Database Cracking. In: CIDR (2007)
26. Ivanova, M., Kersten, M.L., Manegold, S., Kargin, Y.: Data Vaults: Database Technology for Scientific File Repositories. Computing in Science and Engineering **15**(3) (2013)
27. Jensen, A.H., Lauridsen, F., Zardbani, F., Idreos, S., Karras, P.: Revisiting multidimensional adaptive indexing. In: EDBT (2021)
28. Jugel, U., Jerzak, Z., Hackenbroich, G., Markl, V.: VDDa: Automatic Visualization-driven Data Aggregation in Relational Databases. VLDBJ (2015)
29. Kalinin, A., Çetintemel, U., Zdonik, S.B.: Interactive Data Exploration Using Semantic Windows. In: SIGMOD (2014)
30. Karpathiotakis, M., Alagiannis, I., Ailamaki, A.: Fast Queries Over Heterogeneous Data Through Engine Customization. PVLDB **9**(12) (2016)
31. Karpathiotakis, M., Branco, M., Alagiannis, I., Ailamaki, A.: Adaptive Query Processing on Raw Data. PVLDB **7**(12) (2014)
32. de Lara Pahins, C.A., Stephens, S.A., Scheidegger, C., Comba, J.L.D.: Hashedcubes: Simple, Low Memory, Real-time Visual Exploration of Big Data. TVCG **23**(1) (2017)
33. Lins L.D., Klosowski J.T., Scheidegger C.E.: Nanocubes for real-time exploration of spatiotemporal datasets. TVCG **19**, (2013)
34. Liu, C., Wu, C., Shao, H., Yuan, X.: Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. TVCG **26**(1) (2020)
35. Maroulis, S., Bikakis, N., Papastefanatos, G., Vassiliadis, P.: RawVis: A System for Efficient In-situ Visual Analytics. In: SIGMOD (2021)
36. Maroulis, S., Bikakis, N., Papastefanatos, G., Vassiliadis, P., Vassiliou, Y.: Adaptive indexing for in-situ visual exploration and analytics. In: DOLAP (2021)
37. Miranda, F., Lins, L., Klosowski, J.T., Silva, C.T.: TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data. TVCG **24**, 1394–1407 (2017)
38. Morton, K., Balazinska, M., Grossman, D., Mackinlay, J.D.: Support the Data Enthusiast: Challenges for Next-generation Data-analysis Systems. PVLDB **7**(6) (2014)
39. Nathan, V., Ding, J., Alizadeh, M., Kraska, T.: Learning multi-dimensional indexes. In: SIGMOD (2020)
40. Nerone, M., Holanda, P., de Almeida, E.C., Manegold, S.: Multidimensional Adaptive and Progressive Indexes. In: ICDE (2021)
41. Olma, M., Karpathiotakis, M., Alagiannis, I., Athanassoulis, M., Ailamaki, A.: Slalom: Coasting through Raw Data Via Adaptive Partitioning and Indexing. PVLDB **10**(10) (2017)
42. Olma, M., Karpathiotakis, M., Alagiannis, I., Athanassoulis, M., Ailamaki, A.: Adaptive partitioning and indexing for in situ query processing. VLDBJ (2019)
43. Pavlovic, M., Sidlauskas, D., Heinis, T., Ailamaki, A.: QUASII: query-aware spatial incremental index. In: EDBT (2018)
44. Rahman, P., Jiang, L., Nandi, A.: Evaluating Interactive Data Systems. VLDBJ **29**(1) (2020)
45. Rahman, S., Aliakbarpour, M., Kong, H., Blais, E., Karahalios, K., Parameswaran, A.G., Rubinfeld, R.: I've Seen "Enough": Incrementally Improving Visualizations to Support Rapid Decision Making. PVLDB **10**(11) (2017)
46. Richter, S., Quiané-Ruiz, J., Schuh, S., Dittrich, J.: Towards zero-overhead static and adaptive indexing in Hadoop. VLDBJ **23**(3) (2014)
47. Tauheed, F., Heinis, T., Schürmann, F., Markram, H., Ailamaki, A.: SCOUT: Prefetching for Latent Feature Following Queries. PVLDB **5**(11) (2012)
48. Tian, Y., Alagiannis, I., Liarou, E., Ailamaki, A., Michiardi, P., Vukolic, M.: Dinodb: An Interactive-speed Query Engine for Ad-hoc Queries on Temporary Data. IEEE Trans. on Big Data (2017)

49. Wang, Z., Ferreira, N., Wei, Y., Bhaskar, A.S., Scheidegger, C.: Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. TVCG **23**(1) (2017)
50. Wasay, A., Wei, X., Dayan, N., Idreos, S.: Data Canopy: Accelerating Exploratory Statistical Analysis. In: SIGMOD (2017)
51. Yesilmurat, S., Isler, V.: Retrospective adaptive prefetching for interactive Web GIS applications. GeoInformatica **16**(3) (2012)
52. Zardbani, F., Afshani, P., Karras, P.: Revisiting the theory and practice of database cracking. In: EDBT (2020)
53. Zhao, W., Rusu, F., Dong, B., Wu, K., Ho, A.Y.Q., Nugent, P.: Distributed caching for processing raw arrays. In: SSDBM (2018)