Adaptive Indexing for In-situ Visual Exploration and Analytics

Stavros Maroulis
Nat. Techn. Univ. of Athens &
ATHENA Research Center
Greece

Nikos Bikakis
ATHENA Research Center
Greece

George Papastefanatos
ATHENA Research Center
Greece

Panos Vassiliadis
University of Ioannina
Greece

Yannis Vassiliou
Nat. Techn. Univ. of Athens
Greece

ABSTRACT

In-situ processing has received a great deal of attention in recent years. In in-situ scenarios, big raw 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 present an adaptive indexing scheme that enables efficient visual exploration and analytics over big raw data files. Beyond visual exploration and statistics, the scheme enables categorical-based analytics using group-by and filter operations. The proposed scheme combines a *tile-based structure* that offers efficient exploratory operations over the 2D space, with a *tree-based structure* that organizes a tile's objects based on their categorical values, enabling efficient visual analytics and the support of advanced visualization methods. The index 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. We conduct experiments using real and synthetic datasets, and demonstrate that the proposed approach, is in most cases more than 40× faster compared to the existing solutions, and performs around 3 orders of magnitude less I/O operations.

KEYWORDS

Big Raw Data, Visualization, Visual Analytics, Interactive and Progressive Indexing, Categorical Attributes, RawVis

1 INTRODUCTION

Commonly, in data exploration scenarios, users wish to *visually interact and analyze* large data files that *do not fit in main memory*, e.g., data produced by scientific workflows, IoT devices or crowdsourcing. These users usually have limited skills in data management and processing as well as *limited resources or commodity hardware for use*, in contrast to, e.g., a distributed environment. Ideally, the tasks in such scenarios, require a very small *raw data-to-analysis time* and *memory resources*, as well as *efficient visual exploration and analytic operations*, which are performed via interactive visualizations, such as maps, scatter plots, histograms, or other analytical and statistical methods, e.g., OLAP analysis, correlation, clustering, and regression.

Consider the following real-word example, which refers to a common task in telecommunication industry. The data scientists working in telco companies analyze network data in order to get insights regarding the network quality of the company. Such data are stored in comma-separated data files and contain signal 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 ($o_1 \sim o_5$), where each of them *represents a signal measurement*. Basically, each entry contains data regarding the: *geographic location* (Lat, Long), *signal strength* (Signal) and *network bandwidth* (Width), as well the categorical characteristics: *brand*, *network provider*, and *network technology* (Net).[2]

Figure 1(c) outlines our working scenario. Assume that a data scientist wishes to *visually explore* the network data using a 2D visualization technique, e.g., scatter plot, map; and *analyze* it using *visual analytics and statistics*. The user first selects the input file and a map as the underlying visualization layout. The file is parsed and an initial "crude" version of the index is constructed Ⓐ. Then, the user interacts and performs visual and analytic operations on the map Ⓑ.

For example, 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, the user may *move* (e.g., pan left) the visualized region in order to explore a nearby area; or *zoom-in/out* to explore a part of the region or a larger area, respectively. For the visualized points, the user wishes to compute *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 proceeds with the analysis of the data based on one or more *categorical attributes*; e.g., generate: (1) a *heatmap* to present the average signal strength per provider and network technology, or (2) a *bar chart* to present the average signal strength for each provider, or (3) *parallel coordinates* to present the number of measures grouped by provider, brand, and network technology Ⓓ.

Eventually, each user interaction is mapped to a query evaluated over the index Ⓔ and triggers the readjustment of the index structure and the update of its contents Ⓕ.

In the last few years, the *in-situ paradigm* has been adopted in the context of data exploration scenarios, referring to data access methods that enable the analysis over large sets of raw data, i.e., data files in raw formats like CSV or JSON. In-situ techniques attempt to avoid the overhead of fully loading and indexing the data in a DBMS and improve performance by progressively building an index as the user explores data.

Recent works in this area have proposed techniques for progressive loading and indexing of the data, for generic in-situ querying (mainly range queries) [6, 20, 25, 26, 33, 34, 39], and for 2D visual operations over numeric attributes [11, 12]. However, in the context of in-situ processing, no attention has been given, to exploratory aggregate queries, on data with categorical attributes and specifically, *group-by queries* that *filter* and *aggregate* results based on taxonomies and controlled vocabularies, which comprise the domains of the categorical attributes.

As demonstrated in the motivating example, the *Group-by operation is essential in order to generate the most-known visualization types*, in which categorical-based aggregated results are visualized. Such visualization types include: bar charts, heatmaps, parallel coordinates, (binned) scatter plots, radar chart, pies, etc. The great

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

[2] For simplicity, the numbers shown do not correspond to real values; also, Samsg stands for Samsung and Veriz for Verizon.

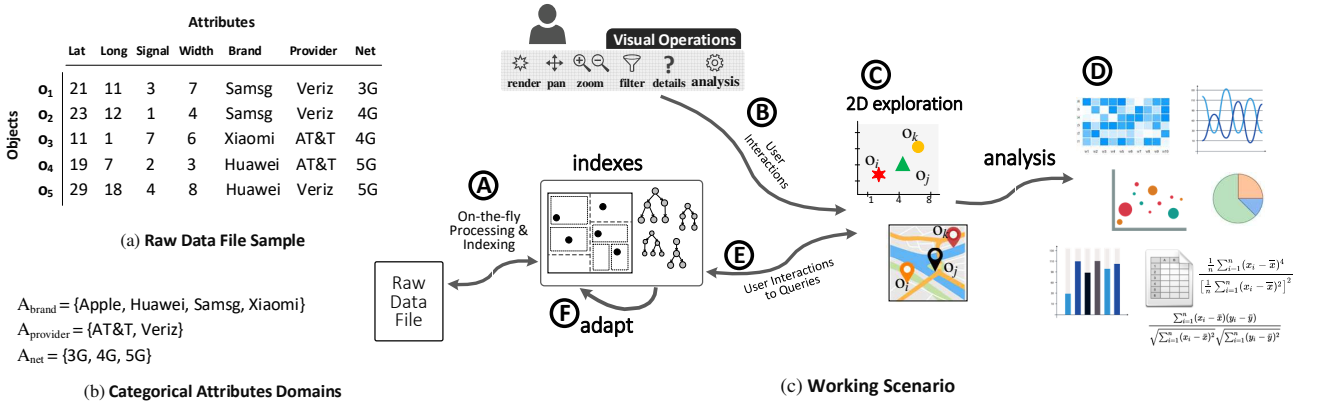


Figure 1: Working Scenario Overview

importance of categorical-based visualization types in data analysis, is also verified by [31] showing that bar charts are by far the most commonly used visualization type. Beyond the group-by operation, the *Filter operation over categorical attributes, enables the support of effective exploration mechanisms*, such as faceted search.

In this work, we propose an innovative *indexing scheme* and *adaptive techniques* in the context of in-situ visual exploration, which supports efficient categorical-based *group-by* and *filter* operations, combined with 2D *visual interactions*, such as exploration of data points on maps. To the best of our knowledge, there is no work studying categorical-based operations in in-situ scenarios. The proposed scheme employs a *tile-based structure* which offers efficient exploration over the 2D plane, with a *tree-based structure* that organizes tile's objects based on its categorical values. The index 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. In our experiments we illustrate that the proposed method *in most cases is about* 40× *faster and performs up to* 3 *orders of magnitude less I/O operations*, compared to existing solutions.

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

- We formulate exploratory and analytical operations over categorical attributes (i.e., group-by, filter, and aggregate), that are mapped to query operators and evaluated over the underlying indexing scheme.
- We design a main-memory lightweight tree structure that organizes objects and computes statistics based on categorical attributes.
- We introduce a hybrid index that combines tile and tree structures. The index organizes the objects based on two dimensions, as well as based on the categorical attributes' values of the objects.
- We design interaction-based adaptation techniques that progressively adjust the index structure based on the user interaction.
- We evaluate the performance of our method in terms of execution time and I/O operations, using real and synthetic datasets. We show that the proposed approach outperforms competitors both in execution time and I/O operations.

Compared to our previous work in the context of in-situ visual exploration [12], this work enables the support of categorical attributes, which are not previously considered. In this context, we define three categorical-based operations (Sect. 2), a tree-based structure (Sect. 3) which is integrated with the tile-based structure proposed in [12] (Sect. 4); and initialization (Sect. 4.3), query processing (Sect. 5.1), and adaptation methods (Sect. 5.2) for the introduced index and operations. The methods have been encapsulated into the *RawVis system* [29] offering efficient in-situ visual analytics.

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. Then, Section 5 presents the query evaluation and the adaptation techniques. Section 6 presents the experimental evaluation, and Section 7 the related work. Finally, Section 8 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 over categorical attributes. Particularly, we define the group-by, filter, and aggregate operations.

Raw Data File & Objects. We assume a *raw data file* $\mathcal{F}$ containing a set of *d-dimensional objects* O. Each dimension corresponds to an *attribute* $A \in \mathcal{A}$, where A may be spatial, numeric, categorical, or textual. Each object o_i is defined as a list of d attribute values $o_i = (a_{i,1}, a_{i,2}, ..., a_{i,d})$, and it is associated with an *offset* f_i (a hex value) pointing to the "position" of its first attribute from the beginning of the file $\mathcal{F}$. Also, given an object o_i and an attribute A, as $a_{i,A}$ we denote the A value of 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).

Given a raw data file, the user arbitrarily selects 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 user selects to 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 a set of *visible objects* $O_\Phi \subseteq O$, for which the values of their axis attributes fall within the ranges of that area.[4]

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

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

[4] In order to express the first query, we assume that the user knows the min/max values of the axis attributes. Otherwise, these values are determined by parsing the file once.

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., attribute 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.

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 O and the axis attributes A_x and A_y, an *exploratory query* Q over O is defined by the tuple $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{G}, \mathsf{N} \rangle$, 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 $\mathsf{S} = (I_x, I_y)$ with its intervals to be referred as $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. This clause, selects the objects $O_\mathsf{S} \subseteq O$ for which both of their axis attributes have values within the respective intervals; i.e., their axis attributes' values are included in the 2D area (i.e., plane) specified by the intervals $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$. The *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 O_S, returning the objects 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 (that satisfy the filter), 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 $A_i \in C$, which are used in a group-by operation. Given an set of objects O and a set of attributes C, the *group-by operation* partitions O into a set of distinct groups, denoted as $\mathcal{G}_O^C$, based on the different combinations of the values of the 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 O_Q, resulting in the groups $\mathcal{G}_{O_Q}^\mathsf{G}$.

Analysis clause L: defines two sets of algebraic aggregate functions (e.g., count, sum, mean), 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 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).[5]

Intuitively, the Selection and Filter clauses apply restrictions to the entire space of objects, resulting in a set of qualifying objects O_Q, which is visually presented. For each object in O_Q, the values of the attributes included in the Details clause will be returned. Then, the Group-by clause evaluates group-by operations over the O_Q objects. Finally, the set of aggregate functions of the Analysis clause are computed over the objects of 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*.

Query Result. The *result $\mathcal{R}$ of an exploratory query Q* over 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 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,
$\mathcal{V}_{x,y,\mathsf{D}} = \{\langle o_i : \alpha_{i,x}, \alpha_{i,y}, \quad \alpha_{i,A_1}, ...\alpha_{i,A_k} \rangle, \forall o_i \in O_Q\}$, where $\{A_1, ...A_k\} = \mathsf{D}$.
(2) $\mathcal{V}_{L_Q}$ is a list of the numeric values resulted from computing the aggregate functions L_Q over the objects O_Q returned by the query. Formally, $\mathcal{V}_{L_Q} = \{\ell_1(O_Q), \ell_2(O_Q), ...\ell_k(O_Q)\}, \forall \ell_i \in L_Q$.
(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 group g_i of the group-by resulted groups $\mathcal{G}_{O_Q}^\mathsf{G}$. The tuple of a group g_i 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,
$\mathcal{V}_\mathsf{G} = \{\langle g_i : a_{i,A_1}, ...a_{i,A_k}, \quad \ell_1(g_i), ...\ell_z(g_i) \rangle, \forall g_i \in \mathcal{G}_{O_Q}^\mathsf{G}\}$, where $\{A_1, ...A_k\} = \mathsf{G}$ and $\{\ell_1, ...\ell_z\} = L_\mathsf{G}$.

3 CATEGORICAL-BASED TREE INDEX

In this section, we present a *tree structure that organizes objects based on its categorical attribute* values, named CET (Categorical Exploration Tree).[6] 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.

Considering the number of attribute value combinations which are required for categorical indexing, a significant amount of memory is required. Hence, the design of a *memory-efficient categorical structure* is a major challenge, especially in our scenario, where we consider limited available resources. To this end, in CET, each object is stored (once) in the leaves, and allocates only three numeric values (i.e., two axis attributes and a file offset). Further, statistics are also only stored in the leaves, since the hierarchical structure of CET allows the efficient computation of statistics over different levels, by performing efficient, in-memory aggregate operations. Note that in our implementation categorical attribute values are mapped to distinct numeric value.

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, we are able to *access the raw file in a sequential manner*. The sequential file scan increases the number of I/Os over continuous disk blocks and improves the utilization of the look-ahead disk cache (more details in Sect. 5.1).

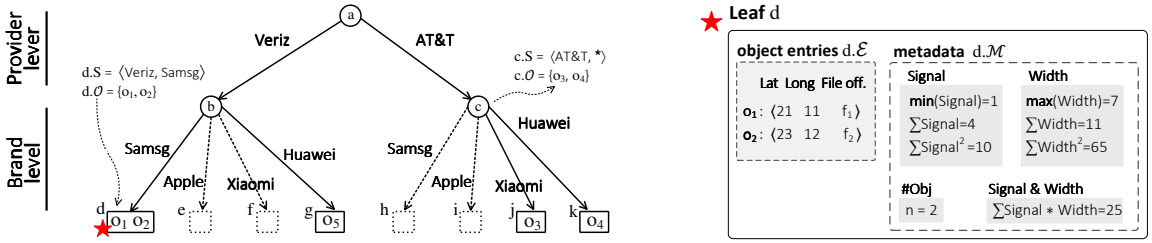
3.1 The CET Tree

In this section, we present the basic concepts of the CET tree. Given a set of objects O and a list of categorical attributes $C = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$, a CET tree h organizes the objects $h.O$ based on the values of the categorical attributes $h.C$.

The *height* of h is $|C|$, so it has $|C| + 1$ *levels* (from 0 to $|C|$), with the *leaf nodes* storing the objects.

[5] More than 90% and 75% of the statistics supported by SciPy [4] and Wolfram [5], respectively, are defined as algebraic aggregate functions [41].

[6] CET is also referred to as *tree*.



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

Figure 2: CET Tree Overview

The CET follows a "level-based" organization, where *each level corresponds to a different attribute*. Based on the given order of the attributes C, the *nodes at level i have edges* that correspond to a different value of the attribute $A_{C_i} \in C$, 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 $|C|$ values, where the value v_i corresponds to a value of the attribute at 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 $|C| - i$ 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* $n.O \in O$, where its attribute values 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.

Object Entries. *Leaf nodes* contain references to the data objects, i.e., object entries. For each object $o_i \in n.O$, an *object entry* e_i is defined as $\langle a_{i,x}, a_{i,y}, f_i \rangle$, with $a_{i,x}, a_{i,y}$ being 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 file (e.g., a long). The file offset f_i defines a "direct and precise" 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 over one or more attributes of *all* $n.\mathcal{E}$ objects such as *sum*, *mean*, *sum of squares of deltas*, etc. Using the leaves' 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.

Example 1. [*CET Tree*] Figure 2a presents the CET index constructed for the categorical attributes $C = \{A_{Provider}, A_{Brand}\}$. The dotted lines denote parts of the tree that are not going to be constructed for this dataset.

Considering the level-based organization, the level 0 corresponds to the *Provider* attribute (the first attribute in 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}.

Additionally, the node c has the *associated sequence values* $c.S = \langle$AT&T, $\star\rangle$, 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* $c.O = \{o_3, o_4\}$ 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 $d.O = \{o_1, o_2\}$ that matches its values $d.S = \langle$Veriz, Samsg$\rangle$ (Fig. 2b). Here, metadata stores statistics regarding the *Signal* and the *Width* numeric attributes. ■

3.2 Tree Operations

In this section, we define the basic operations over the CET tree and we study its space complexity.

Insert Object & Construct Operation. The **insertion** operation inserts an object o into a CET tree h. It takes as input, a tree h, the object o, and a list of categorical attributes $C = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$ based on which the tree organizes its objects. The the tree construction *operation* implemented via an *insert* for each object defined in these operations.

In brief, an object o is inserted to the corresponding leaf l. In order to find l, we need to traverse the tree starting from the root; during the traversal, nodes and edges which do not exist in the tree, are constructed. The path that indicates l is defined by the values of o in C attributes.

Get Leaves/Objects Based on Filter Conditions. Here we present the **get leaves/objects** operation *under filter conditions*. The operation returns the leaf nodes $\mathcal{L}$ of a tree h, that are matched based on the categorical conditions defined in the Filter clause F of a query. The operation, based on the conditions, constructs a path expression p starting from the root to the leaf nodes. Then, for each path produced by the path expression, it traverses the tree in a top-down fashion. The union of the leaves $\mathcal{L}$ reached by the paths is returned. The *get objects based on filter condition* operation is implemented by returning the object's entries of the leaves $\mathcal{L}$.

Space Complexity. 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 no appear in the data objects are not inserted in the tree structure. As a result, the number of nodes depends not only on the attributes and its domain, but also on the (distinct) values of the attributes in the data objects.

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 $h.C = \{A_{C_0}, A_{C_1}, ...A_{C_k}\}$, the *maximum number of nodes* $h.N$ in the worst case is: $h.N = |\text{dom}(A_{C_0})| + |\text{dom}(A_{C_0})| \cdot |\text{dom}(A_{C_1})| + ... + |\text{dom}(A_{C_0})| \cdot |\text{dom}(A_{C_1})| \cdot ... \cdot |\text{dom}(A_{C_k})| = \sum_{i=0}^{k-1} \prod_{j=0}^{i} |dom(A_{C_j})|$. Also, each object entry is contained in only one leaf, we have that the space of all tree objects is $O(|h.O|)$. Hence, the *space complexity* of a CET tree h is: $O(\sum_{i=0}^{k-1} \prod_{j=0}^{i} |dom(A_{C_j})| + |h.O|)$.

4 THE VETI INDEX

In this section, we present the proposed indexing scheme, that combines tile structures with trees.

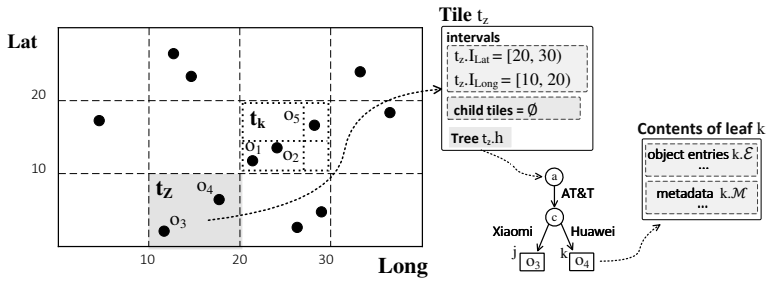


Figure 3: VETI Index Overview

4.1 Tile-based Structure

Our work is built on top of the VALINOR index [12], referred also as *tile-structure*. VALINOR is employed as the underlying indexing technique 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. The basic concepts of the index are:

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

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* $t.O$, where the axis values $a_{i,x}$ and $a_{i,y}$ of each object $o_i \in t.O$ 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, with each leaf tile being associated with a CET tree.

Given a raw data file $\mathcal{F}$, two axis attributes A_x and A_y, and a set 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.

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 $t.h$. In analogy, a tree h *is associated with a leaf tile* t. A tile's enclosed objects $t.O$ are stored in the leaf nodes of the associated tree $t.h$, as objects entries. In case the objects of a tile are not indexed based on the categorical attributes, the tree h corresponds to a node (root) that stores all the object entries.

The VETI *index* $\mathbb{I}$ is defined by a tuple $\langle \mathsf{IP}, \mathsf{AP} \rangle$, IP is the *initialization policy* defining the methods that determine the characteristics of VETI; AP is the *adaptation policy* defining the methods for reconstructing the tiles and trees based on user interaction.

As *basic operations* of the VETI we consider: **initialization** (Sect. 4.3), **query evaluation** (Sect. 5.1), and **adaptation** (Sect. 5.2).

Example 2. [***VETI Index***] Figure 3 presents the VETI index that results by the running example data (Fig. 1). VETI divides the 2D space into 4×3 equally sized disjoint tiles, and the 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 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 $t_z.C$, and a pointer to its tree $t_z.h$. Finally, the tree that corresponds to the tile t_j and the content of the leaf node k are shown in the figure (the objects entries and metadata details are omitted in this figure). ■

4.3 VETI Initialization Overview

In our scenario, we do not consider any loading or preprocessing. The index is constructed following the first user interaction. During the initialization phase the following tasks are realized. First, 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. The algorithm takes as input, the raw file $\mathcal{F}$, the axis and categorical attributes A_x, A_y and $\mathcal{A}_C$ and the first exploratory query Q_0; and returns the initialized index $\mathbb{I}$ and the results $\mathcal{R}_0$ of the Q_0.

Initially, we have to determine the characteristics of the tile structure. The determTiles method (*line* 1) defined by the initialization policy IP, determines the tile structure (i.e., number, size and intervals of the tiles). Next, based on the defined tiles' characteristics, a flat tile structure $\mathbb{I}_\mathcal{T}$ is constructed, i.e., the intervals of each tile are stored.

In the next part (*loop in line* 3), the algorithm scans the file $\mathcal{F}$. For each object o_i, the algorithm reads the attribute values of axis attributes $a_{i,x}$, $a_{i,y}$, the values for its categorical attributes, and the attribute values which are required to evaluate the Analysis clause of Q_0 (*line* 4). Next, the tile t_i that encloses o_i is found (*line* 6), and the insertToTree method (Sect.3.2), inserts o_i into the CET tree $t_i.h$ (*line* 7) of t_i. 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.

Determine Tile Structure. The determTiles method (*line* 1) defined by the tile initialization policy IP, determines the tile structure (i.e., number, size and intervals of the tiles). These characteristics can be defined via numerous approaches (e.g., given by the user or determined by the visualization setting, resolution). Here, we use a locality-based probabilistic initialization approach [12], that is based on the first user query. The main idea of this method is based on the locality-based characteristics of exploration scenarios, in which queries are highly likely to reside in areas near the previous ones [8, 24, 42]. Hence, tiles near the first user query have higher probability to overlap with a future query, compared to tiles farther away. In order to improve the query evaluation performance, the index structure has to reduce the number of I/O operations during query processing. Recall that, in VETI, aggregate metadata are stored in the tree of each tile. This way, queries can exploit the tree metadata of the tiles that overlap, and avoid I/O operations. The use of metadata depends on whether an overlapping tile is fully or partially-contained in the query's window (in 2D space). Apparently, the metadata of a

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

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

1 $\mathbb{I}_{\mathcal{T}} \leftarrow$ IP.determTiles(A_x, A_y, Q_0) //find the number, size & intervals of the tiles
2 $\mathbb{I}_{\mathcal{T}} \leftarrow$ constructTiles($\mathbb{I}$) //construct index tiles $\mathbb{I}_{\mathcal{T}}$
3 **foreach** $o_i \in \mathcal{F}$ **do** //read objects from file, insert them to trees & evaluate Q_0
4 read from $\mathcal{F}$ the values of axis and categorical C, and the attributes required to evaluate the Q_0 Analysis clause
5 use the o_i attributes to evaluate Q_0
6 $t_i \leftarrow$ find the tile $t_i \in \mathbb{I}_{\mathcal{T}}$ that encloses o_i based on its axis attributes values
7 insertToTree($t_i.h, C, o_i$) //insert o_i to tree $t_i.h$
8 **return** $\mathbb{I}$, $\mathcal{R}_0$

Algorithm 2. Query Evaluation ($\mathbb{I}$, Q, $\mathcal{F}$)

Input: $\mathbb{I}$: index (initialized); Q ⟨S, F, D, G, L⟩: query; $\mathcal{F}$: raw data file
Variables: $\mathcal{T}_S$: leaf tiles that overlap with the Selection clause (i.e., 2D area); $\mathcal{T}_a$: tiles resulted by adaptation; $\mathcal{L}$: tree leaf nodes selected by the Query;
Parameters: AP: adaptation policy;
Output: $\mathcal{R}$: result of query Q

1 $\mathcal{L} \leftarrow \emptyset$
2 $\mathcal{T}_S \leftarrow$ getOverlappingLeafTiles ($\mathbb{I}_{\mathcal{T}}$, S)
3 **foreach** $t_s \in \mathcal{T}_S$ **do**
4 $\mathcal{T}_a \leftarrow$ AP.adaptTileAndTree (t_s, Q) //see Sect. 5.2
5 $\forall t_a \in \mathcal{T}_a$: $\mathcal{L} \leftarrow \mathcal{L} \bigcup$ getLeavesBasedOnFilter ($t_a.h$, F)
6 $\mathcal{W}(\langle l, \mathcal{V} \rangle) \leftarrow$ getLeavesRequiringFileAccess ($\mathcal{L}, Q$) //Set of tuples, where $\mathcal{V}$ are the (objects') attributes of the leaf l where their values have to be retrieved from the file
7 **if** $\mathcal{W} \neq \emptyset$ **then** //values are missing — read from file
8 *read from file* the values of attributes $\mathcal{V}$ for each leaf $l \in \mathcal{W}$
9 updateLeafMetadata (l) $\forall l \in \mathcal{W}$
10 $\mathcal{R} \leftarrow$ *evaluate* Q using the objects and the metadata of leaves $\mathcal{L}$
11 **return** $\mathcal{R}$

fully-contained tile could be used, without the need to access the file, while in partially-contained cases, the file has to be accessed in order to compute the metadata referred to the query's contained part. More details about query evaluation over VETI are presented in Section 5.1.

Our initialization method defines a tile structure that is more fine-grained (i.e., having a large 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 gets bigger. Increasing the number (i.e., decreasing the size) of tiles near the first query, increases the possibility that subsequent user queries in this neighborhood overlap with fully-contained tiles, which in turn reduces the I/O cost (for more details see [12]).

Remark. There are cases where the user's exploration is performed in more than one sessions. In such scenarios, the user can perform the exploration, store and reload intermediate states of the index in subsequent user sessions. Thus, the user avoids the cost of re-initialization, as well as the fine tuning (e.g., adaptation) of the index to the previously performed user's operations.

5 QUERY PROCESSING & INDEX ADAPTATION

This section describes the evaluation of the queries and the index adaptation.

5.1 Query Processing

An overview of the query evaluation is presented in Algorithm 2. The algorithm takes as input, the initialized index $\mathbb{I}$, an exploratory query Q and the raw file $\mathcal{F}$.

Find and Adapt Query Related Tiles & Trees. Once the index has been initialized, to evaluate a query Q, we need to find the tiles $\mathcal{T}_S$ that overlap with the 2D area defined in the query's Selection clause S (*line* 2). Specifically, function getOverlappingLeafTiles first 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 policy AP, the adapt procedure (Proc. 1), performs the tile splitting and reorganizes the objects in the trees of the tiles $\mathcal{T}_a$ created by the splitting process (more details in Sect. 5.2).

Then, considering any conditions over categorical attributes that are defined in the Filter clause, getLeavesBasedOnFilter (Sect. 3.2) retrieves the leaf nodes $\mathcal{L}$ of the affected trees (*line* 5). In other words, $\mathcal{L}$ are the leaves of the trees that belong to tiles overlapping the query, after the categorical conditions have been applied in these trees.

Determine the Objects that Require File Access. Procedure getLeavesRequiringFileAccess (O_Q, Q) (*line* 6) determines the objects for which we have to access the raw file in order to answer the query.

To this end, we need to consider the spatial relation between the 2D area specified in the Selection clause and the tiles it overlaps. Specifically, a tile that overlaps a 2D window query can be *partially-contained* or *fully-contained* in it. For a fully-contained tile, we need not examine its objects in order to find the ones that are included in the window. Also, metadata that refers to the objects of a tile can be utilized to avoid accessing the raw file.

For each leaf node in $\mathcal{L}$, procedure getLeavesRequiringFileAccess first checks if the tile it belongs to is partially or fully-contained in the Selection clause of the query. For a node that belongs to a partially-contained tile we need to retrieve from the file the attributes included in the Analysis clause for every one of its objects that is contained in the window query. In contrast, if a node belongs to a fully-contained tile, we know that all of its objects are contained in the Selection clause of the query. File access is required if a Details clause is requested in the query, or no metadata exist for that node to be used to evaluate the Analysis Clause. Also, in case the tile has not been initialized with a CET tree and the query includes a Filter or Group-by clause on some of the categorical attributes, we need to access the raw file to read these attributes for all the objects of that tile.

Next, we access the file for the objects contained in leaves $\mathcal{L}_m$ and retrieve in memory the missing attributes C_m (line 8). Note that if a leaf node belongs to a partially-contained tile, we only access the file for its objects that are included in the window query. To improve the performance of reading the missing attributes from file, 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 objects file offset. This sequential file reading results in faster I/O operation by utilizing the look-ahead disk cache.

Then, based on the values read from the raw file, function updateLeafMetadata computes and updates the metadata of the corresponding leaf nodes, considering the aggregate functions that are used in the Analysis clauses of the query.

Evaluate Query. Finally, we evaluate query Q using the objects $O_{\mathcal{L}}$ and metadata of the leaf nodes $\mathcal{L}$ (*line* 10). The evaluation of the Filter clause of the query starts implicitly when determining the set of leaf nodes using function getLeavesBasedOnFilter based on the filter conditions over the categorical attributes. Here, we use the attribute values $\mathcal{V}$ retrieved from the file to check

Procedure 1: adaptTileAndTree(t, Q)

Input: t: leaf tile to adapt; Q: query
Parameters: AP: adaptation policy
Output: $\mathcal{T}_a$: tiles resulted from t after adaptation

```
1 if AP.splitRequired (t, Q) then
2     T_a ← AP.splitTile (t)
3     AP.generateTreesInSplittedTiles (t.h, T_a, Q)
4 else
5     T_a ← t
6 return T_a
```

every object in $O_{\mathcal{L}}$ against the filter conditions that do not involve the categorical attributes. Finally, for the evaluation of the Group-by and Analysis clauses, we utilize existing metadata for nodes belonging to fully-contained tiles with trees indexing every categorical attribute included in the Filter and Group-by clauses. For all other cases, we evaluate the functions requested in the Analysis clause using the values retrieved from the file.

5.2 Incremental Adaptation

VETI employs an *incremental index adaptation* technique that attempts to adapt the index structure to the query workflow of the user exploration. The adaptation in VETI may incrementally split a tile that overlaps the Selection clause, into smaller subtiles. This tile splitting increases the likelihood that a future query will fully overlap a tile in the area that the user exploration focuses, and will improve query performance by reducing accesses to the file.

Procedure adapt (Proc. 1) 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 the window query if it needs to be split. To determine if a tile requires (further) splitting, splitRequired function (*line* 1) estimates the number of I/O operations needed to evaluate query Q in a tile t and compares it to a numeric threshold for the maximum number of I/Os. If the expected I/O cost for a tile exceeds that threshold, a split is performed. To estimate the number of necessary operations, we take into consideration not just the Selection clause but also the Filter clause. Specifically, even if the objects belonging to a tile may exceed the threshold set, query Q may filter on a categorical attribute and using the tile's CET tree the number of objects we need to examine may be much less than the threshold. Details about the splitting model are presented in [12].

Tile Splitting. The split is performed in function splitTile (line 2) which returns a new set of tiles $\mathcal{T}_a$. Each one of the child tiles that result 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].

6 EXPERIMENTAL ANALYSIS

6.1 Experimental Setup

Datasets. We have used a *real dataset*, the *NYC Yellow Taxi Trip Records* (TAXI), which is a CSV file, containing information regarding yellow taxi rides in NYC[7]. From this dataset, we selected a subset that includes taxi trip records in 2014 (165M objects, 26 GB) with each record object referring to a specific taxi ride described by 18 *attributes* (e.g., pick-up and drop-off dates and locations, trip distances, fares, and tip amount). From the TAXI dataset, we selected the pickup location longitude and latitude as the axis attributes of the exploration, while the tip amount was selected as the attribute in the Analysis clause, for which aggregates were evaluated. The TAXI dataset includes 5 categorical attributes (e.g. payment type, passenger count, and rate type) with cardinality varying between 2 and 9. Each query is defined over an area of 500m × 500m size (i.e., window size), simulating a map-based exploration at the neighborhood zoom level, with the first query Q_0 posed in central Manhattan. Each query contains a Group-by clause on the passenger count attribute, while the Filter clause includes 1 to 2 filter conditions randomly specified over the remaining categorical attributes.

Regarding the *synthetic datasets* (SYNTH10/50), we have generated two files of 100M *data objects*, having 10 and 50 *attributes* (11 and 51 GB, respectively). The synthetic datasets contain numeric attributes in the range (0, 1000), as well as 6 categorical attributes. The default cardinality for the categorical attributes is 10, while we also generated three variations of the SYNTH10 dataset where the cardinality of every categorical attribute was 20 and 50 respectively. All attributes in the synthetic datasets follow a uniform distribution. In our experiments, we selected two of the numeric attributes as the axis attributes of a 2D exploration scenario, and another one was selected as the attribute for which aggregates were evaluated. For the query sequences we generated for the synthetic dataset, we used a window size with approximately 100K objects.

Evaluation Scenarios. We study the following visual exploration scenario: (1) First, the user selects the two axis attributes and requests to explore a region of the data from the raw file, specifying also an attribute for which the aggregate functions (avg, sum, min, max and standard deviation) will be evaluated, as well as selecting a set of categorical attributes that are of interest for the exploration. For this action, referred to as "From-Raw Data-to-1stResults", we measure the *execution time* for creating the index and answering the first query, the results of which are evaluated directly on the raw file, during index initialization. (2) Next, the user continues exploring neighboring areas, while also filtering the data and analyzing it based on values for a Group-by attribute. For this, we generated sequences of 100 overlapping queries, with each window query shifted in relation to its previous one by 1-20% towards a random direction. This scenario attempts to formulate a common user?s behavior in 2D visual exploration, where the user explores nearby regions using pan operations [8, 24, 42]. For example, assume the common "region-of-interest" or "following-a path" scenarios in map visual exploration.

Further, each query contains a Group-by clause over a single categorical attribute for the Group-by clause, while we randomly alternated the Filter clause to include 1 or 2 equality conditions over the remaining categorical attributes of the dataset. To reflect our exploration-oriented assumption that attributes included in the initial query Q_0 are more likely to be included in the next queries, we generated the query sequences so that these attributes appeared more frequently in the Filter clause of the queries.

VETI Parameters. The VETI's tile structure was 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 locality-based initialization method [12]. The numeric threshold for the adaptation of VETI was set to 200.

Index Initialization Budget. In our experiments, we assume a *categorical-based index initialization budget*, which is an upper bound of the memory required for constructing the CET trees of the tiles. This budget includes only the memory allocated by the tree structures, and does not include the memory required to store the object entries. The tree memory cost is mainly determined by the number of tree nodes, which is in turn defined by the attributes and their domains. In our estimation for the tree cost, we assume

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

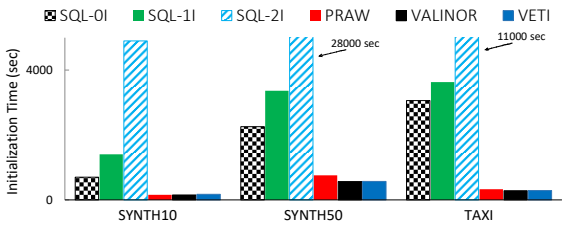


Figure 4: Initialization Time (Time includes: File Parsing, Index Construction & Q_0 Evaluation)

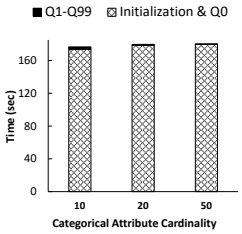


Figure 5: Initiation & Execution Time: Varying Cardinality of Categorical Attributes [SYNTH10]

that all distinct values of an attribute appear in the objects of a tile and, as a result, in the tree to be constructed. Based on the initialization budget and the estimated cost of the CET tree, we sort tiles based on their distance from Q_0 and assign trees to them until the initialization budget is exhausted. For the remaining tiles, we do not construct tree structures during initialization. The CET tree of such a tile will be constructed on demand, should a future query overlap it. In our experiments, the index initialization budget was set to 2GB.

Competitors. We compare our method with: (1) VALINOR [12] which contains only the tile-based indexing structure without the CET tree structure for supporting efficient evaluation of queries with categorical attributes; (2) A popular DBMS (MySQL 8.0.22), in which the entire dataset is loaded and indexed in advance; three indexing settings are considered: (*a*) no indexing (SQL-0I); (*b*) one composite B-tree on the two axis attributes (SQL-1I); and (*c*) two single B-trees, one for each of the two axis attributes (SQL-2I). MySQL also supports SQL querying over external files (see CSV Storage Engine in Sect. 7); however, due to low performance [6], we do not consider it as a competitor in our evaluation. (3) PostgresRaw (PRAW)[8], build on top of Postgres 9.0.0 [6], which is a generic platform for in-situ querying over raw data (Sect. 7).

Metrics. In our experiments, we measure the: (1) *execution time* for each query; (2) *accumulative execution time* for the entire exploration scenario; and (3) the number of *I/O operations*.

Implementation. We have implemented VETI on JVM 1.8 and the experiments were performed on an 3.60GHz Intel Core i7 with 64GB of RAM. We applied memory constraints (12GB max Java heap size); however, PRAW required more than 32GB and 50GB for the synthetic and the TAXI dataset, respectively.

6.2 Results

Initialization Time. In this experiment, we measure the time required to answer the first query Q_0. This time also includes the time required for initializing each one of the methods we examine. Besides executing Q_0, MySQL needs to load and index the data, while PRAW needs to parse the raw file and construct the positional map. When evaluating Q_0, VALINOR generates the tile index structure and populates it with the object entries, while VETI needs to also construct the categorical-based tree indexes.

Figure 4 presents the results for every dataset used. MySQL exhibits the worst performance for evaluating the first query. MySQL needs to parse all attributes of the raw file and store all data on disk. Also, for the SQL-1I and SQL-2I cases, the corresponding indexes must be built, which explains the increased initialization time in relation to SQL-0I where no index is generated.

Both VALINOR and VETI 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.

VETI is slightly slower during the initialization compared to VALINOR. This is expected as VETI needs to also determine the tile-tree assignments, parse the categorical attributes for all objects and create the corresponding tree structures. Despite this slight difference in initialization time, VETI performs much faster than VALINOR when answering queries that include the categorical attributes.

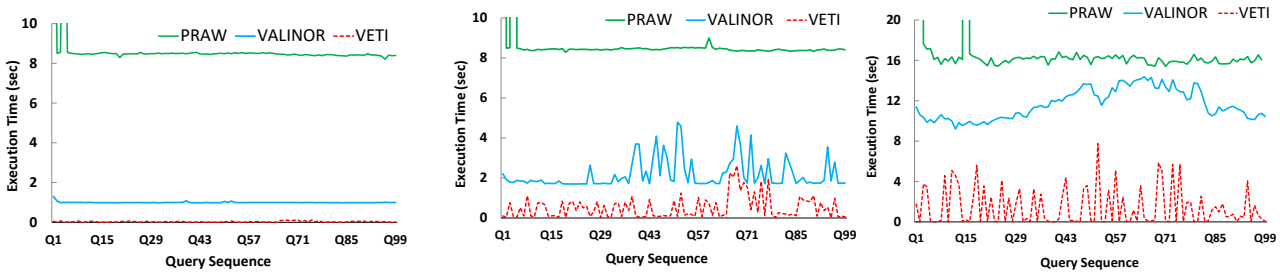
Modify the Cardinality of Categorical Attributes. For this experiment, we study the effect of the cardinality of the categorical attributes in the performance of VETI. We ran this experiment against 3 versions of the SYNTH10 dataset where the cardinality of the categorical attributes for each one was set to 10, 20 and 50 respectively. The cumulative time needed to execute the complete query sequence of the exploration scenario is shown in Figure 5. It is the time needed to execute the complete workload by VETI, including Q_0 which is depicted separately from all subsequent queries as it represents the initialization time of the index. Observe that the initialization time increases slightly with increasing cardinality. This increase can be attributed to the higher construction cost of the wider CET trees when the cardinality of the categorical attributes increases. On the contrary, the total execution time of queries $Q_1 \sim Q_{99}$ decreases with higher cardinality (2.8, 1.4, 0.8 for cardinality 10, 20 and 50 respectively). This decrease in execution time is directly related to the number of I/O operations we need to perform during the workload. The objects of the synthetic dataset have values uniformly distributed over each attribute's domain. As a result, in the case of a higher cardinality for the categorical attributes, the same number of filter conditions are satisfied by a smaller number of objects, requiring fewer overall I/O operations.

Performance during the Exploration Scenario. In the next experiment, we compare the query evaluation performance of VETI against the competitors. The execution time for queries $Q_1 \sim Q_{99}$ is shown in Figure 6. Note that Q_0 is omitted in this figure, as it includes the initialization time which was examined separately in Figure 4. In the results, we omit the plots for MySQL as it exhibits a much higher execution time.

Compared to the other methods, VETI exhibits significantly lower execution time in almost all cases. Regarding PRAW, we observe that it performs much worse than both VALINOR and VETI for all datasets examined. 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 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. Observe that some of the early queries in the sequence exhibit execution time significantly higher than the rest, and comparable to the time required to answer Q_0. Since the queries we issue have 1 or 2 randomly specified filter conditions on the categorical attributes, these queries need to tokenize and parse attributes that were not included in Q_0, and populate the positional map with these. This explains their higher execution time. Once the positional map has been populated with all the attributes including in the queries, PRAW exhibits a relatively constant execution time.

Compared to VALINOR, as we can observe, VETI exhibits a much faster execution time for every query in the workload.

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



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

Figure 6: Execution Time for Entire Exploration Scenario ($Q_0 \sim Q_{99}$)

Even though both indexes attempt to adapt to the workload and maintain metadata to speed up query execution time by reducing I/Os, VALINOR does not include any indexing capabilities for categorical attributes. This results in not being able to utilize tile-based aggregate metadata to evaluate queries that include a Group-by clause or a Filter clause that refers to a categorical attribute. In contrast, VETI organizes object entries and metadata in tile-tree structures based also on the values for the categorical attributes. For this reason, when a query includes a categorical attribute, VETI traverses the tree structures of every overlapping tile and filters the objects it needs to retrieve from the raw file. Also, if any leaf tree nodes of a fully-contained tile contain aggregate metadata, it can be utilized to avoid accessing the raw file.

The execution time for VETI as well as for VALINOR is mainly determined by the number of rows that need to be retrieved from the raw file. This can be seen in Figure 7, where the I/O plots exhibit approximately the same behavior as the corresponding execution time plots in Figure 6. Compared to VALINOR, VETI requires fewer I/O operations for every query in the sequence. VALINOR has to access the raw file for every object contained in the 2D window query since it has to retrieve their values for the categorical attributes specified in the Filter and Group-by clause.

7 RELATED WORK

In-situ Processing. Data loading and indexing usually take a large part of the overall time-to-analysis for both traditional RDBMs and Big Data systems [20]. In-situ query processing aims at avoiding data loading in a DBMS by accessing and operating directly over raw data files. NoDB [6] is a philosophy for constructing a no-dbms querying architecture, and PostgresRAW is one of the first efforts for in-situ query processing. PostgresRAW incrementally builds on-the-fly auxiliary indexing structures called "positional maps" which store the file positions of data attributes, as well as it stores previously accessed data into cache. As opposed to 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 with filter conditions on categorical attributes.

DiNoDB [39] is a distributed version of PostgresRAW. In the same direction, RAW [26] extends the positional maps in order to both index and query files in formats other than CSV. In the same context, Proteus [25] supports various data models and formats. Recently, Slalom [33, 34] exploits the positional maps and integrates partitioning techniques that take into account user access patterns.

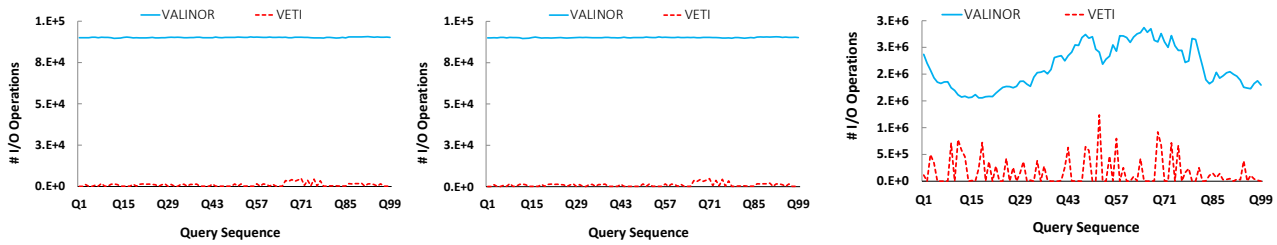
RawVis [11, 12] defines a tile-based index in the context of in-situ visual exploration, supporting 2D visual operations over numeric attributes. Compared to VETI, RawVis does not support operations and indexing over categorical attributes. As a result, it cannot exploit well-know visualization techniques, such as bar charts, heatmaps, pies and parallel coordinates. Particularly, VETI extends a tile-based structure similar to RawVis, with trees that enrich tiles with information about categorical attributes.

Additionally, 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 PostgreSQL has the Foreign Data [3]. However, these tools do not focus on user interaction, parsing the entire file for every query, and resulting in significantly low query performance for interactive scenarios [6]. The aforementioned works study the generic in-situ querying problem without focusing on the specific needs for raw data visualization and exploration. In addition, due to low query performance cannot be used in exploration scenarios. Instead, our work considers the in-situ processing of a specific query class, that enables user operations and visual analytics. The goal of our solution is to optimize these operations, so 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 [15] 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 [27], Hashedcubes [14], SmartCube [28], Gaussian Cubes [40], and TopKubes [30] 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 [27] attempts to reduce the memory of the data cube by sharing nodes in a single tree structure. Hashedcubes [14] 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 [28] 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 do not address the need of reducing the initialization time, i.e., they cannot be used in in-situ scenarios. Moreover, a major difference compared to our approach, is that these works assume that all the aggregations are materialized and stored in memory, which in the case of very fine-grained spatial indexing or many categorical dimensions, can require prohibitive amounts of memory. In contrast, our approach adapts the granularity of the spatial index based on user interaction.

Further, graphVizdb [9, 10] 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.



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

Figure 7: Number of I/O Operations for the Entire Exploration Scenario ($Q_0 \sim Q_{99}$)

In a different context, tile-based structures are used in visual exploration scenarios. Semantic Windows [24] 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 [16] 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 [7, 17–19, 21–23, 32, 37, 38], is to incrementally build and adapt indexes during query processing, following the characteristics of the workload. However, in these works the data has to be previously loaded in the system, i.e., a preprocessing phase is required. As a result, these approaches are not suitable for in-situ query scenarios, where the cost of the preprocessing phase has to be minimized. In addition, the existing cracking and adaptive indexing methods have been developed in the context of column-stores [7, 17–19, 21–23, 37], or MapReduce systems [38]. On the other hand, VETI has been developed to handle raw data stored in text files with commodity hardware. Finally, [35, 36] study the problem of incremental indexing for 3D spatial data.

8 CONCLUSIONS

In this paper, we have presented an indexing scheme and adaptive processing methods for in-situ visual exploration and analysis that allow the user to combine visual exploration of data from a raw file on a 2D canvas with sophisticated analysis over its categorical attributes. This scheme combines tile structures with trees and is progressively adapted based on user interaction. Finally, the presented method was evaluated experimentally.

Acknowledgment. This work is 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: Open Source Scientific Tools for Python. http://www.scipy.org.
[5] Wolfram : Descriptive Statistics. https://reference.wolfram.com/language/tutorial/DescriptiveStatistics.html.
[6] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.
[7] K. Alexiou, D. Kossmann, and P. Larson. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB*, 6(14), 2013.
[8] L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD*, 2016.
[9] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis. Towards Scalable Visual Exploration of Very Large Rdf Graphs. In *ESWC*, 2015.
[10] N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis. Graphvizdb: A Scalable Platform for Interactive Large Graph Visualization. In *ICDE*, 2016.
[11] N. Bikakis, S. Maroulis, G. Papastefanatos, and P. Vassiliadis. RawVis: Visual Exploration over Raw Data. In *ADBIS*, 2018.
[12] N. Bikakis, S. Maroulis, G. Papastefanatos, and P. Vassiliadis. In-situ Visual Exploration over Big Raw Data. *Information Systems*, 95, 2021.
[13] N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis. A Hierarchical Aggregation Framework for Efficient Multilevel Visual Exploration and Analysis. *SWJ*, 2017.
[14] C. A. de Lara Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, Low Memory, Real-time Visual Exploration of Big Data. *IEEE TVCG*, 23(1), 2017.
[15] M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. Vistrees: Fast Indexes for Interactive Data Exploration. In *HILD*, 2016.
[16] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2), 1998.
[17] G. Graefe and H. A. Kuno. Self-selecting, self-tuning, incrementally optimized indexes. In *EDBT*, 2010.
[18] F. Halim, S. Idreos, P. Karras, and R. H. C. Yap. Stochastic Database Cracking: Towards Robust Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 5(6), 2012.
[19] P. Holanda, S. Manegold, H. Mühleisen, and M. Raasveldt. Progressive Indexes: Indexing for Interactive Data Analysis. *PVLDB*, 12(13), 2019.
[20] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here Are My Data Files. Here Are My Queries. Where Are My Results? In *CIDR*, 2011.
[21] S. Idreos, M. L. Kersten, and S. Manegold. Database Cracking. In *CIDR*, 2007.
[22] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, 2009.
[23] S. Idreos, S. Manegold, H. A. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4(9), 2011.
[24] A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive Data Exploration Using Semantic Windows. In *SIGMOD*, 2014.
[25] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12), 2016.
[26] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on Raw Data. *PVLDB*, 7(12), 2014.
[27] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for Real-Time Exploration of Spatiotemporal Datasets. *IEEE TVCG*, 19:2456–2465, 2013.
[28] C. Liu, C. Wu, H. Shao, and X. Yuan. Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE TVCG*, 26(1), 2020.
[29] S. Maroulis, N. Bikakis, G. Papastefanatos, and P. Vassiliadis. RawVis: A System for Efficient In-situ Visual Analytics. *SIGMOD*, 2021.
[30] F. Miranda, L. Lins, J. T. Klosowski, and C. T. Silva. TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data. *IEEE TVCG*, 2017.
[31] K. Morton, M. Balazinska, D. Grossman, and J. D. Mackinlay. Support the Data Enthusiast: Challenges for Next-generation Data-analysis Systems. *PVLDB*, 7(6), 2014.
[32] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska. Learning multi-dimensional indexes. In *SIGMOD*, 2020.
[33] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through Raw Data Via Adaptive Partitioning and Indexing. *PVLDB*, 2017.
[34] M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Adaptive partitioning and indexing for in situ query processing. *VLDBJ*, 2019.
[35] M. Pavlovic, D. Sidlauskas, T. Heinis, and A. Ailamaki. QUASII: query-aware spatial incremental index. In *EDBT*, 2018.
[36] M. Pavlovic, E. Tzirita Zacharatou, D. Sidlauskas, T. Heinis, and A. Ailamaki. Space odyssey: efficient exploration of scientific data. In *ExploreDB*, 2016.
[37] E. Petraki, S. Idreos, and S. Manegold. Holistic Indexing in Main-memory Column-stores. In *SIGMOD*, 2015.
[38] S. Richter, J. Quiané-Ruiz, S. Schuh, and J. Dittrich. Towards zero-overhead static and adaptive indexing in Hadoop. *VLDBJ*, 23(3), 2014.
[39] Y. Tian, I. Alagiannis, E. Liarou, A. Ailamaki, P. Michiardi, and M. Vukolic. Dinodb: An Interactive-speed Query Engine for Ad-hoc Queries on Temporary Data. *IEEE Transactions on Big Data*, 2017.
[40] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *IEEE TVCG*, 23(1), 2017.
[41] A. Wasay, X. Wei, N. Dayan, and S. Idreos. Data Canopy: Accelerating Exploratory Statistical Analysis. In *SIGMOD*, 2017.
[42] S. Yesilmurat and V. Isler. Retrospective adaptive prefetching for interactive Web GIS applications. *GeoInformatica*, 16(3), 2012.