RawVis: Visual Exploration over Raw Data

Nikos Bikakis[0000−0001−6859−1941], Stavros Maroulis, George Papastefanatos[0000−0002−9273−9843], and Panos Vassiliadis[0000−0003−0085−6776]

University of Ioannina, Greece

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

Keywords: In situ query, Big raw data, Adaptive processing, Visual analytics, Visualization, Indexing, User Interaction, Exploratory data analysis

1 Introduction

In situ data exploration [1, 15–17] is a recent trend in raw data management, which aims at enabling on-the-fly scalable querying over large sets of volatile raw data, by avoiding the loading overhead of traditional DBMS techniques. A common scenario is that users wish to have a quick overview, explore and analyze the contents of a raw data file through a 2D visualization technique (e.g., scatter plot, map).

As an example, a scientist (e.g., astronomer) wishes to visually explore and analyze sky observations stored in raw data files (e.g., csv) using the Sloan Digital Sky Survey (SDSS) dataset (www.sdss.org), which describes hundreds of millions of sky objects (e.g., stars). First, the scientist selects the file and visualizes a part of it using a scatter plot with the sky coordinates (e.g., right ascension and declination). Then, she may focus on a sky region (e.g., defining coordinates and area size), for which all contained sky objects are $rendered$; $move$ (e.g., pan left) the visualized region in order to explore a nearby area; or $zoom\text{-}in/out$ to explore a part of the region or a larger area, respectively. She may also click on a single or a set of sky objects and view $details$, such as *name* and *diameter*; $filter$ out objects based on a specific characteristic, e.g., *diameter* larger than 50 km; or $analyze$ data considering all the points in the visualized region, e.g., compute the average *age* of the visualized objects.

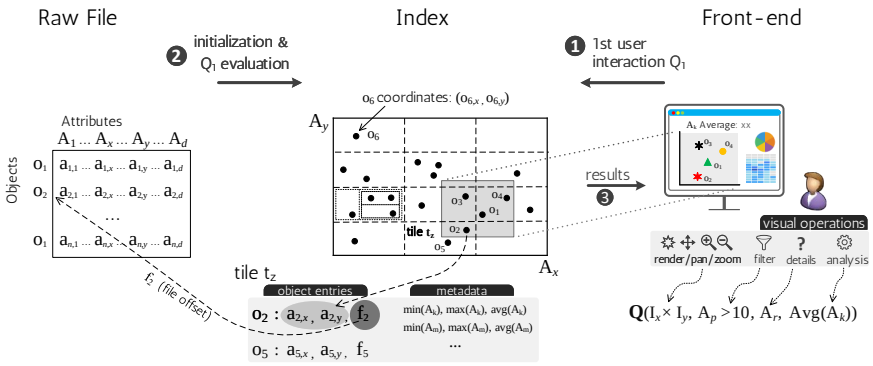

Fig. 1: RawVis Framework Overview

Most experimental and commercial visualization tools perform well for ad-hoc visualizations of small files (e.g., showing a trend-line or a bar chart) or over aggregated data (e.g., summaries of data points, into which user can zoom in), which *can fit in main memory*. For larger files, these tools usually require a preprocessing step for data to be *loaded* and either *indexed* in a traditional database, or *distributedly stored* and queried by *non-commodity hardware* (e.g., a Big Data tool).

On the other hand, in the in situ exploration scenarios, large data files which *do not fit in main memory*, must be efficiently handled *on-the-fly* using *commodity hardware* [1, 15–17]. In the in situ visual exploration scenarios, several challenges arise.

A first requirement is that no offline preprocessing is to be performed and any preprocessing, such as indexing or repartitioning must be performed on-the-fly, minimizing the user involvement and time overhead. Secondly, a non-expert user with limited programming or scripting skills must be supported to access and analyze data through visual ways, i.e., via an intuitive set of visual rather than data-access (e.g., querying) operations. Further, the response time of such visual operations must be significantly small (e.g., less than 1sec) in order to be acceptable by the user. Finally, the aforementioned operations have to be performed in machines with limited computational, memory and space resources, i.e., using commodity hardware.

In this work, we address the aforementioned challenges for enabling interactive 2D visual exploration scenarios of large raw data files using commodity hardware. We present the RawVis framework, which is built on top of a lightweight main memory index, VALINOR (Visual AnaLysis Index On Raw data), constructed on-the-fly given the first user query and adapted based on the user interaction.

In our working scenario (Figure 1), we assume that a user visually explores multi-dimensional objects stored in a raw file, using a 2D visualization technique. The user initially selects two attributes (A_x and A_y) as the X and Y axis of the visualization. In the first user interaction, the entire raw file is parsed and an "abstracted" version of the VALINOR is built on-the-fly, organizing the data objects into *tiles* based on their A_x and A_y values. Further, the index stores auxiliary metadata in each tile regarding its contents (e.g., average attribute values), in order to reduce computation cost and access to the raw file. Throughout the exploration, visual user operations (e.g., pan, zoom, filter) are expressed as queries evaluated over VALINOR. Following the user interaction,

VALINOR incrementally reorganizes the objects' grouping, constructs tile hierarchies, and recomputes and enriches metadata.

The main contributions of this work are: (1) we formulate visual user interactions as data-access operations; (2) we design an index in the context of in situ visual exploration over large data; (3) we describe the query processing techniques over this index; (3) we conduct an experimental evaluation using real and synthetic datasets, as well as several systems and structures; i.e., MySQL, PostgresRaw, and R-tree, which shows that our technique outperforms competitors both in execution time and memory consumption.

2 Preliminaries

Data File & Objects. We assume a *raw data file* $\mathcal{F}$ containing a set of d*-dimensional objects* $\mathcal{O}$. Each dimension j corresponds to an *attribute* $A_j \in \mathcal{A}$, where each attribute may be numeric or textual. Each object o_i contains a list of d attributes $o_i = (a_{i,1}, a_{i,2}, ..., a_{i,d})$, and it is associated with an *offset* f_i in $\mathcal{F}$ pointing to the "position" of its first attribute, i.e., $a_{i,1}$. Note that, object entries can be either fixed or variable-length, in the latter case they are separated by a special-character; e.g., CR for a text file, that precedes the offsets.

Visual Exploration Scenario. Given a set of d-dimensional objects, the user arbitrarily selects two attributes $A_x, A_y \in \mathcal{A}$, whose values are numeric and can be mapped to the X and Y axis of a 2D visualization layout. A_x and A_y attributes are denoted as *axis attributes*, while the rest as *non-axis*.

The user follows a sequence or combination of the following *visual operations* to interact with the data: (1) $render$: visualizes the objects included in a specified 2D area, denoted as *visualized area*. (2) $move$: changes the visualized area (i.e., pan). (3) $zoom\ in/out$: specifies a new visualized area that is within (resp. covers) the current visualized area. (4) $filter$: excludes objects from the visualized area, based on conditions over non-axis attributes. (5) $details$: presents values of non-axis attributes. (6) $analyze$: analyzes data from the objects included in the visualized area.

Note that, as previously illustrated, multiple visual operations can be performed in a single user interaction; e.g, zoom in a region while filtering the presented objects.

Exploratory Query. Considering the aforementioned visual operations, we define the corresponding data-access operators. In what follows, we formulate the notion of an exploratory query. Given a set of d-dimensional objects $\mathcal{O}$ and the axis attributes A_x and A_y, an *exploratory query* Q over $\mathcal{O}$ is defined by the tuple $\langle \mathsf{S}, \mathsf{F}, \mathsf{D}, \mathsf{N} \rangle$, where:

– The *Select part* S defines a 2D range query (i.e., window query) specified by two intervals $\mathsf{S}.I_x$ and $\mathsf{S}.I_y$ over the axis attributes A_x and A_y, respectively. This part selects the objects $\mathcal{O}_\mathsf{S} \subseteq \mathcal{O}$ for each of which both of their axis attributes have values within the respective intervals, defined by the window query. The select part is *mandatory*, while the rest parts are *optional*.

– The *Filter part* F defines conditions over the non-axis attributes. As $\mathcal{A}_\mathsf{F}$ we denote the set of attributes involved in F. In our current implementation, the conditions in F are expressed using a single attribute, unary and binary arithmetic operations, and

constants. The filter part is applied over the objects $\mathcal{O}_\mathsf{S}$, selecting the objects $\mathcal{O}_Q \subseteq \mathcal{O}_\mathsf{S}$ that satisfy F. If the filter part is not defined (i.e., $\mathsf{F} = \varnothing$), then $\mathcal{O}_Q = \mathcal{O}_\mathsf{S}$.

– The *Details part* D defines a set $\mathcal{A}_\mathsf{D}$ of non-axis attributes. The query returns the values of these attributes for each object in $\mathcal{O}_Q$.

– The *Analysis part* N defines aggregate, analytic, or user-defined functions over numeric attributes of the objects $\mathcal{O}_Q$. As $\mathcal{A}_\mathsf{N}$ we denote the attributes that are used in these functions. In our current implementation, we consider a single attribute and the following aggregate functions: count, sum, average, min, and max. The analysis part returns the values $\mathcal{V}_\mathsf{N}$ from the evaluation of the specified functions.

The semantics of query execution involves the evaluation of the four parts in the following order: (1) *Select* part; (2) *Filter* part; (3) *Details*; (4) *Analysis* part.

Query Result. An exploratory query Q returns the axis values for the objects $\mathcal{O}_Q$ along with their values $\mathcal{V}_\mathsf{D}$ of the attributes specified in D, denoted as $\mathcal{O}_Q^\mathsf{D}$; and the numeric values $\mathcal{V}_\mathsf{N}$ resulted from the analytic part. Formally, the result is consisted by: (1) a set of tuples $\mathcal{O}_Q^\mathsf{D} = \langle \alpha_{i,x}, \alpha_{i,y}, \alpha_{i,A_{\mathsf{D}_1}}, ...\alpha_{i,A_{\mathsf{D}_d}} \rangle, \forall o_i \in \mathcal{O}_Q, \forall d \in \{1, ...|\mathcal{A}_\mathsf{D}|\}$ with $A_{\mathsf{D}_d} \in \mathcal{A}_\mathsf{D}$; and (2) a list of numeric values $\mathcal{V}_\mathsf{N}$.

Example (*From Visual operations to Exploratory query*). Each visual operation can be expressed as an exploratory query. Specifically, the $render$ operation is implemented using only the Select part of the query, setting the intervals equal to the values of the visualized area. Assuming our running example, in which A_x = *declination* and A_y = *right ascension*. A $render$ operation that visualizes the rectangle sky area from 100° to 110°, and from 20° to 25°, is executed by defining $\mathsf{S}.I_x$=[100°, 110°] and $\mathsf{S}.I_y$=[20°, 25°].

The $move$, $zoom\ in/out$ operations are also implemented by defining a Select part, having as parameters the coordinates of the neighboring, contained/containing visualized regions, respectively. For example, a $zoom\text{-}out$ operation over the previously presented area is executed as $\mathsf{S}.I_x$=[97.5°, 112.5°] and $\mathsf{S}.I_y$=[18.75°, 26.25°]. Further, the $filter$ operation is implemented by including a Filter part. In our example F = "*diameter* $>$ 50 km". Finally, the $details$ and $analyze$ operations correspond to the Details and Analysis parts. For example the details and analyze operations described in our example correspond to D = {*name*, *diameter*} and N = "*avg(age)*", respectively.

3 The VALINOR Index

The VALINOR is a lightweight *tile-based multilevel* index, which is stored in memory, organizing the data objects of a raw file, into *tiles*. The index is constructed on-the-fly given the first user query and incrementally adjusts its structure to the user visual interactions. In the construction, each tile is defined on a fixed range of values of the A_x and A_y axis attributes, by dividing the euclidean space (defined by the A_x and A_y domains) into initial tiles. Then, user operations split these tiles subsequent into more fine-grained ones, thus forming a hierarchy of tiles. In each level of the hierarchy, all tiles are disjoint (i.e., non-overlapping) and can belong to only one parent tile. Next we formalize the main concepts of the proposed index.

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

Tile. A *tile* t is a part of the Euclidean space defined by two intervals $t.I_x$ and $t.I_y$. Each t is associated with a *set of object entries* $t.\mathcal{E}$, if it is a leaf tile, or a set of *child tiles* $t.C$, if it is a non-leaf tile. The set $t.\mathcal{E}$ is defined as a set of object entities, such that for each $e_i \in t.\mathcal{E}$ its attribute values $a_{i,x}$ and $a_{i,y}$ fall within the intervals of the tile t, $t.I_x$ and $t.I_y$ respectively. Further, t is associated with a set $t.\mathcal{M}$ of *metadata* related with the $t.\mathcal{E}$ objects contained in the tile, e.g., aggregated values over attributes. As $t.\mathcal{M}_A$ we denote the attributes for which metadata has been computed for the tile t.

VALINOR Index. Given a raw data file $\mathcal{F}$ and two axis attributes A_x, A_y, the index organizes the objects into non-overlapping rectangle tiles based on its A_x, A_y values. Specifically, the VALINOR *index* $\mathcal{I}$ is defined by a tuple $\langle \mathcal{T}, \mathsf{IP}, \mathsf{AP}, \mathsf{MH} \rangle$, where $\mathcal{T}$ is the set of *tiles* contained in the index; IP is the *initialization policy* defining the initial tile size; AP is the *adaptation policy* defining a criterion (e.g., the relation of a tile's size w.r.t. the query's window size), and a method for splitting tiles and reorganizing object entries following user's interaction; and MH is the *metadata handler* defining and computing the metadata stored in each tile.

VALINOR Initialization. In our approach we do not consider any preprocessing for the index construction, but rather the index is constructed on-the-fly upon the first time the user requests to visualize a part of the raw file. The file is scanned once, for creating the initial VALINOR structure and computing the results of the first query. The initial version of VALINOR corresponds to a flat tile structure that does neither exhibit any hierarchy nor contain any metadata to the tiles. This phase, referred to as *initialization phase*, aims at minimizing the response time of the first user action, by avoiding computations which may not be used through exploration.

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

Input: $\mathcal{F}$: raw data file; A_x, A_y: X and Y axis attributes;
Parameters: IP: initialization policy
Output: $\mathcal{T}$: initialized index tiles;
$\mathcal{O}_Q^{\mathsf{D}}$, $\mathcal{V}_{\mathsf{N}}$: first query results

1 $\mathcal{T} \leftarrow \varnothing$ // set of tiles
2 $x_0, y_0 \leftarrow \mathsf{IP}.\texttt{getInitialTileSize}()$ // initial tile size
3 **foreach** $o_i \in \mathcal{F}$ **do** // read objects from raw file
4 read $a_{i,x}, a_{i,x}$ from $\mathcal{F}$
5 set $f_i \leftarrow$ offset of $a_{i,1}$ in $\mathcal{F}$
6 append $\langle a_{i,x}, a_{i,y}, f_i \rangle$ to tile $t.\mathcal{E}$, where $t \in \mathcal{T}$ determined from $a_{i,x}, a_{i,y}$ and x_0, y_0
7 compute first query result (O_Q^{D}, $\mathcal{V}_{\mathsf{N}}$) through file parsing
8 **return** $\mathcal{T}$, O_Q^{D}, $\mathcal{V}_{\mathsf{N}}$

Algorithm 1 describes the initialization phase. The input *initialization policy* IP determines the initial size of the tiles $x_0 \times y_0$ (*line* 1). Algorithm 1 first scans the raw file $\mathcal{F}$ once (*loop in line* 3), reads the values of $a_{i,x}$, $a_{i,y}$ f_i of each object o_i (*line* 4 & 5), and appendsa new object e_i to the entries $t.\mathcal{E}$ of the tile t (*line* 6). It, finally, evaluates whether this entry should be included in the results of the initial query.

The initialization policy IP is a parameter that defines the initial tile size. It can be given either explicitly by the user (e.g., in a map the user defines a default scale of coordinates for the initial visualization), be computed based on data characteristics (e.g., the ranges of the A_x, A_y attributes), or adjusted to the visualization setting considering certain characteristics (e.g., screen size/resolution) [4, 13].

4 Query Processing and Index Incremental Adaptation

This section describes the evaluation of exploratory queries over the proposed index.

Query Processing Workflow. Algorithm 2 outlines the workflow of the query evaluation. Given a query Q, we first look up the index $\mathcal{I}$ and determine the tiles $\mathcal{T}_Q$ that overlap with the query. In addition, we examine their objects and select the ones $\mathcal{O}_\mathsf{S}$ (*line* 1) that are contained in the query window.

Considering $\mathcal{T}_Q$, we determine the tiles $\mathcal{T}_\mathcal{F}$ for which we have to access the raw file in order to answer the query (*line* 3). In this step, we also split the tiles $t \in \mathcal{T}_\mathcal{F}$ based on the adaptation policy AP, creating a new set of tiles W, as explained later in the section. Next, we access the file at each offset of the objects in $\mathcal{T}_\mathcal{F}$ tiles and retrieve in memory the attributes defined in the $details$, $filter$, and $analysis$ parts (*line* 7). We use these values for the metadata handler MH to compute and store metadata in each tile (*line* 8). Finally, we evaluate the filter and analysis parts on the retrieved objects (*lines* 9 & 10). Each different operation of Alg. 2 is described below.

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

Input: $\mathcal{I}$: index; Q: query; $\mathcal{F}$: raw data file
Variables: $\mathcal{O}_\mathsf{S}$: objects selected from select part;
$\mathcal{T}_Q$: tiles that overlapped with the select part;
$\mathcal{T}_\mathcal{F}$: tiles for which file access is required
Parameters: AP: adaptation policy; MH: metadata handler
Output: O^D_Q: objects of the result along with the detail values;
$\mathcal{V}_\mathsf{N}$: analysis values

1 $\mathcal{O}_\mathsf{S}, \mathcal{T}_Q \leftarrow$ `evaluateSelectPart`($\mathcal{I}$,S)
2 **foreach** $t \in \mathcal{T}_Q$ **do**
3 **if** `fileAcessRequired`*(t, Q)* **then**
4 $W \leftarrow$ AP.`splitTile`(t)
5 add W into $\mathcal{T}_\mathcal{F}$
6 **if** $\mathcal{T}_\mathcal{F} \neq \varnothing$ **then**
7 $\mathcal{V}_{\mathcal{A}_\mathsf{F}}, \mathcal{V}_{\mathcal{A}_\mathsf{D}}, \mathcal{V}_{\mathcal{A}_\mathsf{N}}, \leftarrow$ `readFile`($\mathcal{T}_\mathcal{F}$, $\mathcal{F}$, $\mathcal{O}_\mathsf{S}$)
8 MH.`updateMetadata`($\mathcal{T}_\mathcal{F}$, Q, $\mathcal{V}_{\mathcal{A}_\mathsf{F}}$, $\mathcal{V}_{\mathcal{A}_\mathsf{N}}$)
9 $\mathcal{O}^\mathsf{D}_Q \leftarrow$ `evaluateFilterPart` ($\mathcal{O}_S$, F, $\mathcal{V}_{\mathcal{A}_\mathsf{F}}$,$\mathcal{V}_{\mathcal{A}_\mathsf{D}}$)
10 $\mathcal{V}_\mathsf{N} \leftarrow$ `evaluateAnalysisPart`($\mathcal{O}_Q$, $\mathcal{V}_{\mathcal{A}_\mathsf{N}}$)
11 **return** O^D_Q, $\mathcal{V}_\mathsf{N}$

Evaluate Select Part. In order to evaluate the Select part over the index (Alg. 2, *line* 1), we have to identify $\mathcal{O}_\mathsf{S}$ by accessing the leaf tiles $\mathcal{T}_Q$ which overlap with the window query specified in Q. Since window queries can be evaluated directly from the index, the Select part is computed without performing any I/Os.

Determine File Read. Procedure 1 determines whether and for which objects, file reads are required . The check is performed for each tile t independently and distinguishes between the type of operation is requested by the query. If details are requested, we always have to access the file (*line* 1); otherwise for each t we check if the Analysis and Filter parts can be evaluated using the metadata that has been already computed for t (*line* 5), by a previous query.

Procedure 1: fileAcessRequired(t, Q)

Input: t: tile; Q: query;
Output: true / false: file access is required

1 **if** $D \neq \varnothing$ **then** //detail part is included
2 **return** true //access file for the $\mathcal{O}_Q$ objects in t
3 **else if** $F = \varnothing$ **and** $N = \varnothing$ **then** //no filter & analysis parts
4 **return** false
5 **else if** $\mathcal{A}_N \subseteq t.\mathcal{M}_A$ **and** F *can be evaluated from* $t.\mathcal{M}_A$ **then** //filter and/or analysis part is included
6 **return** false
7 **else**
8 **return** true //access file for the $\mathcal{O}_Q$ objects in t

Incremental Computation of Metadata & Query Processing. The metadata is computed incrementally and help improving the performance evaluation of the filter expressions and the functions defined in the analysis part. *Incremental computation* implies

Procedure 2: splitTile(t)

Input: t: tile
Parameters: AP: adaptation policy
Output: $\mathcal{T}_a$: tiles resulted from adaptation

1 **if** AP.`checkCondition` *(t) = reconstruction required* **then**
2 $\quad \mathcal{T}_a \leftarrow$ AP.`reconstruct` (t)
3 **else**
4 $\quad \mathcal{T}_a \leftarrow t$
5 **return** $\mathcal{T}_a$;

Procedure 3: readFile($\mathcal{T}_\mathcal{F}$, $\mathcal{F}$, $\mathcal{O}_S$)

Input: $\mathcal{T}_\mathcal{F}$: tiles for which file access is required;
$\mathcal{F}$: raw data file $\mathcal{O}_S$: data selected from the select part
Output: $\mathcal{V}_{\mathcal{A}_\mathsf{F}}, \mathcal{V}_{\mathcal{A}_\mathsf{D}}, \mathcal{V}_{\mathcal{A}_\mathsf{N}}$, attributes values required for the *filter*, *details* & *analysis* part

1 **forall** o_i *included in tiles* $\mathcal{T}_\mathcal{F}$, *with* $o_i \in \mathcal{O}_S$ **do**
2 $\quad$ access $\mathcal{F}$ at file offset f_i
3 $\quad \mathcal{V}_{\mathcal{A}_{\mathsf{F}_i}}, \mathcal{V}_{\mathcal{A}_{\mathsf{D}_i}}, \mathcal{V}_{\mathcal{A}_{\mathsf{N}_i}} \leftarrow$ read the o_i attributes values that are required for the F, D and N parts
4 $\quad$ insert: $\mathcal{V}_{\mathcal{A}_{\mathsf{F}_i}}$ into $\mathcal{V}_{\mathcal{A}_\mathsf{F}}$; $\mathcal{V}_{\mathcal{A}_{\mathsf{D}_i}}$ into $\mathcal{V}_{\mathcal{A}_\mathsf{D}}$; $\mathcal{V}_{\mathcal{A}_{\mathsf{N}_i}}$ into $\mathcal{V}_{\mathcal{A}_\mathsf{N}}$
5 **return** $\mathcal{V}_{\mathcal{A}_\mathsf{F}}, \mathcal{V}_{\mathcal{A}_\mathsf{D}}, \mathcal{V}_{\mathcal{A}_\mathsf{N}}$

that metadata for a tile is not fully computed during the (initial) index construction, but rather, during user exploration, i.e., the first time a query access a tile.

For example, the Analysis part of a user query requires the aggregate (min, sum, etc.) value of a non-axis attribute that was already computed for the tiles overlapping with the query, from a previous user visit. In this case, we do not need to read the non-axis attribute from the file, as we can aggregate the precomputed values for computing the output value of the query. Further, assume that we have a Filter part, e.g., F="*diameter* $>$ 50 km", evaluated over the objects of a tile t, on which we have stored as metadata the maximum value of the *diameter* if its containing objects, e.g., 60 km. Similarly, we can easily answer the query through a single lookup at the index and avoid processing the objects in t. Currently, we consider aggregate values over attributes of the $t.\mathcal{E}$ objects as metadata – i.e., count, sum, average, min, max.

The metadata handler MH, using the values retrieved from file, for each tile (Algo 2, *line* 8): (1) determines for which attributes to compute metadata; (2) computes metadata; and (3) updates metadata in the tiled accessed by the query. The attributes for which metadata is computed are: the attributes $t.\mathcal{M}_A$ for which metadata has been previously computed in t, as well as the attributes included in the query filter $\mathcal{A}_\mathsf{F}$ and analysis part $\mathcal{A}_\mathsf{N}$.

Incremental Index Adaptation. Procedure 2 reorganizes objects in the index by splitting tiles into smaller ones, based on the adaptation policy AP. It first checks whether a tile t must be split (*checkCondition*, *line* 1) and, if so, places all objects in t into the new tiles T that are resulted from splitting (*line* 2). Note that, a tile's splitting (if required) is performed, each time (i.e., *incrementally*) a query accesses the particular tile (i.e., *adaptively*). The incremental adaptation attempts to maximize the number of tiles which are fully-contained in a query window. For a fully-contained tile t, there is no need to: (1) access the file if the required metadata have been computed for t; and (2) examine the objects in t in order to find the objects that are included in the window. Thus, fully-contained tiles reduce both computation and I/O cost.

In our current implementation, AP defines a numeric *threshold* $h > 0$, defining the relation between the tile and the window size. The *checkCondition* method ($line$ 1) examines if the size of the t is more than $1/h$ times larger than the window size. Then, using the *reconstruct* method ($line$ 2), t is *split* into more tiles constructing a tile hierarchy. The splitting is performed following the method used in either Quadtree or k-d tree.

Read File. We use the file offset stored in object entries, in order to access the file (Procedure 3). For each object we read the attributes values required for the filter, details & analysis part. A crucial issue in our index is to improve the execution time of the queries when file access is required. Exploiting the way that VALINOR constructs and stores

the object entries, we are able to scan the raw file in a sequential manner.The sequential file scan increases the number of I/Os over continuous disk blocks and improves the utilization of the look-ahead disk cache.

During the initialization phase, the object entries are appended into tiles as the file is parsed (Alg. 1, *line* 5). As a result, the object entries in each tile are sorted based on object's file offset. In the query evaluation, we identify the tiles $\mathcal{T}_{\mathcal{F}}$ for which we have to read the file (Alg. 2, *line* 3). Considering the lists of objects entries in $\mathcal{T}_{\mathcal{F}}$, we read the objects from lists following a k-*way* merge, i.e., all objects are sorted on their offset before reading the file. This way, objects values are read by accessing the file in sequential order. Note that, in our experiments, the sequential access results in about 8-times faster I/O operations compared to accessing the file by reading objects on a tile basis (i.e., read the objects of tile w, then read the objects of tile v, etc.).

Evaluate Filter, Details and Analysis Parts. In the general case, the *Filter part* requires access to the file to retrieve the values of the attributes included in the filter conditions. However, there are cases where precomputed values in metadata (e.g., min, max) can be exploited to avoid files access. On the other hand, the *Details part* always requires access to the raw file (Procedure 3), since our index does not keep in memory attribute values other than the two axis attributes. Particularly, for each object o_i in, using the file offset pointers, we retrieve the values $\mathcal{V}_{\mathsf{D}_i}$ of the attributes included in the details part. Finally, the *Analysis part* is first evaluated on existing metadata stored in the index; otherwise it requires access to the values $\mathcal{V}_{\mathcal{A}_\mathsf{N}}$ read from the file.

5 Experimental Analysis

5.1 Experimental Setup

Datasets. We have used a *real dataset*, the "Yahoo! Flickr" (YAHOO), which is a csv file, containing information regarding public Flickr photos/videos[1]. YAHOO contains 100M objects and each object refers to a photo/video described by *25 attributes* (e.g., longitude, latitude, accuracy, tags). We consider a map-based visualization, with the axis attributes being the longitude and latitude of the location where a photo was taken. From this dataset, we select the photos/videos posted in USA region, resulting to 13M *data objects* and a csv file of 7 GB. Regarding the *synthetic datasets* (SYNTH), we have generated csv files of 100M *data objects*, having 2, 10, and 50 *attributes* (2, 11, and 51 GB, respectively), with 10 being the default value. Each attribute value is a real number in the range (0, 1000) and follows a uniform distribution.

Competitors. We have compared with: (1) A traditional DBMS (MySQL 5.5.58), where the user has to load all data in advance in order to execute queries; three indexing settings are considered: (a) no indexing (SQL-0I); (b) one composite B-tree on the two axis attributes (SQL-1I); and (c) two single B-trees, one for each of the two axis attributes (SQL-2I). MySQL also supports SQL querying over external files (see CSV Storage

[1] Available at: https://research.yahoo.com

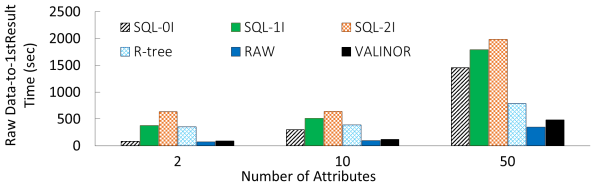

Fig. 2: From-Raw Data-to-1stResult Time (SYNTH dataset)

Engine in Sect. 6); however, due to low performance [1], we do not consider it as a competitor in our evaluation. (2) PostgresRaw (RAW)[2] (build on top of Postgres 9.0.0) [1], which is a generic platform for in situ querying over raw data (see Sect. 6). (3) A main memory Java implementation of the R*-tree[3]. We have tested various configurations for the index fan-out, ranging from 4 to 128; as the difference in the performance is marginal, we only report on the best one, i.e., 16.

Implementation. We have implemented VALINOR[4] in Java and the experiments were conducted on an 2.67GHz Intel Xeon E5640 with 64GB of RAM. We applied memory constraints (max Java heap size) in order to measure the performance of our approach and our competitors in a commodity hardware setting. For large datasets, the available version of RAW, required a significant amount of memory (in some cases more than 32GB); the same held for the in-memory R-Tree implementation (more than 16GB in most cases). In contrast, VALINOR performed well for heap size less than 10GB for the larger dataset of 100M objects, 50 attributes (51 GB).

Evaluation Scenario & Metrics. We study the following visual exploration scenarios: (1) first, the user requests to view a region of the data from the raw file. For this action, referred to as "From-Raw Data-to-1stResult", we measure the time for creating the index and fetching the query results. (2) Next, the user explores neighboring areas, using render, move and zoom operations, denoted as "Basic Visual Operations", for which we measure the query response time. (3) Finally, the user explores neighboring areas of the dataset, requesting also aggregate values for the included objects. For that, we examine the efficiency of our adaptive method, over a sequence of overlapping window queries.

In our experiments, we measured *time*, *memory consumption* and *file accesses* varying the following parameters: *cardinality* (number of objects), *dimensionality* (number of attributes), and *query selectivity* (i.e., objects included in the examined area).

5.2 Results

From-Raw Data-to-1stResult Time. In this experiment, we measured the time required to answer the first query. This corresponds to loading and indexing the data for MySQL, and to constructing the positional map for RAW. For the VALINOR and R-tree cases the in-memory indexes must be built. For the R-tree construction, bulk-loading was used. In VALINOR we used 100 $\times$ 100 tiles for the initialization of the

[2] https://github.com/HBPMedical/PostgresRAW
[3] https://github.com/davidmoten/rtree
[4] The source code is available at https://github.com/Ploigia/RawVis

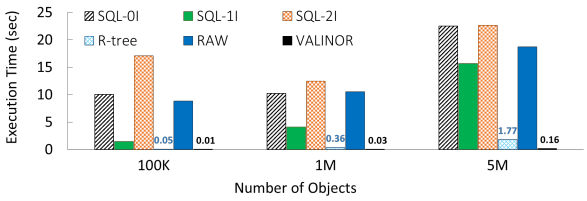

Fig. 3: Execution Time for Basic Visual Operations (SYNTH dataset)

index (this number of tiles is used in all the experiments). Note that, we also examined different number of tiles; however, the effect on the performance was negligible and we do not report on these results. Figure 2 presents the results varying the dimensionality of the objects. VALINOR outperforms the MySQL and R-tree methods, with the difference getting more significant as the dimensionality increases. As expected, VALINOR exhibits a lower initialization time than R*-tree; the latter must determine multilayer MBRs and assign objects to leaf nodes as opposed to our approach which is initialized with fixed tile sizes exhibiting no hierarchy. In this experiment, VALINOR is outperformed by RAW, due to its non-optimized CSV parsing and slower I/O Java operations, as opposed to the efficiency provided by RAW's programming language (i.e., C). This is something we plan to address in the future.

Performance of Basic Visual Operations. In this experiment we study the performance of the $render$, $move$ and $zoom\ in/out$ visual operations. Recall that, these operations are expressed as queries over the axis attributes (i.e., windows queries). We use the SYNTH dataset to evaluate these operations over regions that contain different numbers of objects (100K, 1M, 5M). Figure 3 presents the query execution time. As expected, execution time increases for higher values of selectivity. VALINOR significantly outperforms all methods in all cases.

Regarding RAW, its positional map is used to minimize file parsing and tokenizing costs, which can not be exploited to reduce the number of objects examined in range queries. R-tree is the best competing method. However, its performance is significantly affected by the number of objects. Overall, VALINOR is around 5-12$\times$ faster than R-tree, and more than 1 magnitude faster than the other methods.

Index Adaptation. We define a sequence of neighboring and overlapping queries in order to study the adaptation of the index, and we measure the execution time for each query. To assess the effect of VALINOR's adaptation policy, we compare its performance with that of a "static" VALINOR version (denoted as VALINOR-S), for which tiles are not split as a result of user queries, and we measure the number of objects read from the file for each version. As adaptation policy we consider standard Quadtree splitting of tiles and we only present MySQL-1I, which has exhibited the best performance.

First, we use the *real* YAHOO *dataset* (Fig. 4). We constructed a sequence of ten queries (Q1-Q10), each one defined over an area of 10km $\times$ 10km size (i.e., window size), requesting the average value of one of the non-axis attributes. Every query is *shifted* by 10% of the window size (i.e., 1km) in relation to its previous one, where the shift direction (N, E, S, W) is randomly selected, with the first query Q1 posed in central LA. Note that, we were not able to run RAW in this dataset, due to the types of the attributes included in YAHOO (i.e., textual). For the *synthetic dataset* (Fig. 5), we use

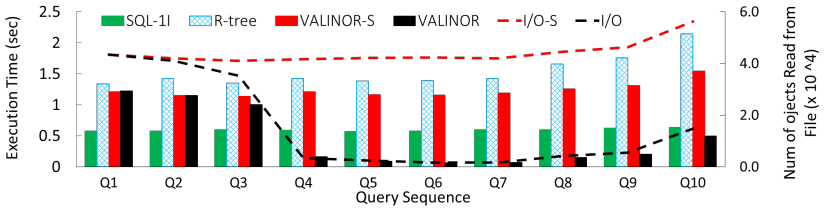

Fig. 4: Execution Time & File Accesses for each Query (YAHOO dataset)

a default window size with approximately 100K objects, and uniform data distribution, shifting each query by 10% of the window size.

Comparing VALINOR with its non-adaptive version, we observe that in both datasets (Fig. 4-5) the adaptive method exhibits better performance in time and file reads. The number of objects read from file is defined in the right axis and depicted with slashed lines. This improvement, as a result of the tile splittings and the computed aggregate values, is obvious even after the first query Q1, with the difference getting more significant after the query Q3. Variations in these improvements are due to the different number of objects contained in each window query, as well as to its position of the query w.r.t. to its previous one. Overall, for the real dataset, the adaptive method requires up to 6× less file reads (slashed lines) and up to 5× less time; 25× and 17× for the synthetic one, respectively.

Compared to the other methods, VALINOR outperforms all methods with the exception of the first 3 queries in the YAHOO dataset (Fig. 4) where SQL performs better. Regarding the RAW method (Fig 5), we observe that it requires approximately the same time for every query, since it does not adapt to the workload and the positional map cannot be exploited to answer the aggregate queries. Considering all the workload (Q1-Q10), in YAHOO, VALINOR requires 4 sec to execute all queries, VALINOR-S 12 sec, R-tree 15 sec, and SQL-1I 6sec; in SYNTH, 74, 82, 105 and 145 sec, respectively.

Memory Consumption. We measured the memory used to build VALINOR and R-tree varying the number of objects in the SYNTH dataset (Fig.6). Note that, the memory consumption in VALINOR and R-tree is not affected by the objects' dimensionality, since in each case, only the two axis attributes are indexed. We did not consider RAW and MySQL settings since they exhibit different memory requirements due to their tight-coupling with the DBMS. We can observe that VALINOR requires significantly less memory than R-tree, with R-tree requiring 2× more memory for 100M objects.

6 Related Work

In situ Query Processing. Data loading and indexing usually take a large part of the overall time-to-analysis for both traditional RDBMs and Big Data systems [11]. In situ query processing aims at avoiding data loading in a DBMS by accessing and operating directly over raw data files. NoDB [1] is one of the first efforts for in situ query processing. NoDB incrementally builds on-the-fly auxiliary indexing structures called "positional maps" which store the file positions of data attributes in order to reduce parsing and tokenization costs during query evaluation, as well as it stores previously accessed

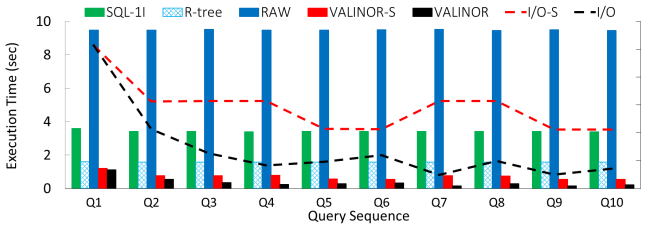

Fig. 5: Execution Time & File Accesses for each Query (SYNTH dataset)

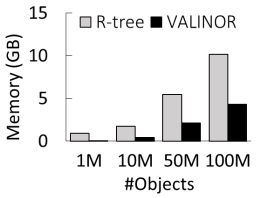

Fig. 6: Memory Consumption (SYNTH dataset)

data into cache. The authors have also developed PostgresRaw, which is an implementation of NoDB over PostgreSQL. DiNoDB [17] is a distributed version of NoDB. In the same direction, RAW [15] extends the positional maps to index and query file formats other than CSV. Recently, Slalom [16] 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 [12] and SDS/Q [5] rely on DBMS technologies to perform analysis over scientific array-based file formats. Further, SCANRAW [6] considers parallel techniques to speed up CPU intensive processing tasks associated with raw data accesses.

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

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

Indexes for Visual Exploration. VisTrees [9] and HETree [4] are tree-based main-memory indexes that address visual exploration use cases; i.e., they offer exploration-oriented features such as incremental index construction and adaptation. Compared to our work, both indexes focus on one-dimensional visualization techniques (e.g., histograms), and they do not consider disk storage; i.e., data is stored in-memory.

Hashedcubes [8] is a main-memory data structure supporting a wide range of interactive visualizations, such as heatmaps, time series, plots. It is based on multiple hierarchical multidimensional (spatial, categorical, temporal) indexes, which are constructed during the loading phase. The construction requires multiple sortings on the input values, which may result in increased amount of time for large datasets. In comparison with our approach, Hashedcubes requires that all data resides in memory, and thus it does not address the need of reducing the overall time-to-visualization (both loading and query processing time) over raw data files and it does not feature any adap-

tive technique based on the user interaction. Further, graphVizdb [3] is a graph-based visualization tool, which employs a 2D spatial index (e.g., R-tree) and maps user interactions into window 2D queries. Compared to our work, graphVizdb requires a loading phase where data is first stored and indexed in a relational database system. In addition, it targets only graph-based visualization and interaction, whereas our approach offers interaction in 2D layouts, such as maps or scatter diagrams.

In different contexts, tile-based structures are used in visual exploration scenarios. Semantic Widows [14] considers the problem of finding rectangular regions (i.e., tiles) with specific aggregate properties in an interactive data exploration scenario. This work uses several techniques (e.g., sampling, adaptive prefetching, data placement) in order to offer interactive online performance. ForeCache [2] considers a client-server architecture in which the user visually explores data from a DBMS. The approach proposes a middle layer which prefetches tiles of data based on user interaction. Prefetching is performed based on strategies that predict next user's movements. Our work considers different problems compared to the aforementioned approaches. However, some of their methods can be exploited in our framework to further improve efficiency.

Traditional Indexes. A vast collection of index structures has been introduced in traditional databases, as well as Big Data systems (see e.g., M-trees [7]). Traditional spatial indexes, such as the R-tree family and kd-trees, are designed to improve the evaluation of a variety of spatial queries and are widely available in both disk-based and main memory implementations. Due to their objective (i.e., support of various spatial query operations), even main memory spatial indexes require substantial memory and time resources to construct [10], which makes them inappropriate for enabling the users to quickly start exploring and interacting with the data, as in the case of in situ data exploration (see also the results in Sect. 5). On the contrary, our approach proposes a main-memory lightweight index, which aims at accelerating the raw data-to-visualization time and offering a simple set of 2D visual operations to the user, rather than covering all aspects of spatial data management.

7 Conclusions

In this paper, we have presented the RawVis framework and the VALINOR index, a lightweight main memory structure, which enables interactive 2D visual exploration scenarios of very large raw data files in the presence of limited memory resources. VALINOR is constructed from a raw data file given the first user query and adapted based on the user interaction. We have formulated a set of simple visual operations and mapped them to query operators evaluated on the VALINOR index. We have conducted a thorough experimental evaluation with real and synthetic datasets and compared with three competitors; i.e., MySQL, PostgresRaw, and R-tree. The results showed that our technique outperforms both in query execution time and memory consumption.

Acknowledgments. This research is implemented through the Operational Program "Human Resources Development, Education and Lifelong Learning" and is co-financed by the European Union (European Social Fund) and Greek national funds.

References

1. I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. Nodb: Efficient Query Execution on Raw Data Files. In *SIGMOD*, 2012.
2. L. Battle, R. Chang, and M. Stonebraker. Dynamic Prefetching of Data Tiles for Interactive Visualization. In *SIGMOD*, 2016.
3. N. Bikakis, J. Liagouris, M. Krommyda, G. Papastefanatos, and T. Sellis. Graphvizdb: A Scalable Platform for Interactive Large Graph Visualization. In *ICDE*, 2016.
4. N. Bikakis, G. Papastefanatos, M. Skourla, and T. Sellis. A Hierarchical Aggregation Framework for Efficient Multilevel Visual Exploration and Analysis. *Semantic Web Journal*, 2017.
5. S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *SIGMOD*, 2014.
6. Y. Cheng and F. Rusu. SCANRAW: a Database Meta-operator for Parallel In-situ Processing and Loading. *ACM Trans. Database Syst.*, 40(3), 2015.
7. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.
8. 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. *TVCG*, 23(1), 2017.
9. M. El-Hindi, Z. Zhao, C. Binnig, and T. Kraska. Vistrees: Fast Indexes for Interactive Data Exploration. In *HILDA*, 2016.
10. S. Hwang, K. Kwon, S. K. Cha, and B. S. Lee. Performance Evaluation of Main-memory R-tree Variants. In *SSTD*, 2003.
11. 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.
12. M. Ivanova, M. L. Kersten, S. Manegold, and Y. Kargin. Data Vaults: Database Technology for Scientific File Repositories. *Computing in Science and Engineering*, 15(3), 2013.
13. U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. VDDa: Automatic Visualization-driven Data Aggregation in Relational Databases. *VLDBJ*, 2015.
14. A. Kalinin, U. Çetintemel, and S. B. Zdonik. Interactive Data Exploration Using Semantic Windows. In *SIGMOD*, 2014.
15. M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on Raw Data. *PVLDB*, 7(12), 2014.
16. M. Olma, M. Karpathiotakis, I. Alagiannis, M. Athanassoulis, and A. Ailamaki. Slalom: Coasting through Raw Data Via Adaptive Partitioning and Indexing. *PVLDB*, 10(10), 2017.
17. 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 TBD*, 2017.