VisTrees: Fast Indexes for Interactive Data Exploration

Muhammad El-Hindi, Zheguang Zhao, Carsten Binnig, Tim Kraska
Brown University
firstname_lastname@brown.edu

ABSTRACT

Visualizations are arguably the most important tool to explore, understand and convey facts about data. As part of interactive data exploration, visualizations might be used to quickly skim through the data and look for patterns. Unfortunately, database systems are not designed to efficiently support these workloads. As a result, visualizations often take very long to produce, creating a significant barrier to interactive data analysis.

In this paper, we focus on the interactive computation of histograms for data exploration. To address this issue, we present a novel multi-dimensional index structure called *VisTree*. As a key contribution, this paper presents several techniques to better align the design of multi-dimensional indexes with the needs of visualization tools for data exploration. Our experiments show that the *VisTree* achieves a speed increase of up to three orders of magnitude compared to traditional multi-dimensional indexes and enables an interactive speed of below 500ms even on large data sets.

1. INTRODUCTION

Motivation: Interactive visualizations are arguably the most important tool to explore, understand and convey facts about data. For example, as part of data exploration, visualizations are used to quickly skim through the data and look for patterns [9, 2]. This requires to generate a sequence of visualizations and to allow the user to interact with them. Figure 1 shows an example screen-shot of an interactive data exploration session over the Titanic data set[1] using PanoramicData [11]. First, the user analyzes the distribution of passengers by region (leftmost) in a type of a choropleth map visualization, then he looks at the histogram of the age of passengers (topmost). Afterwards, he selects only those passengers that are from certain countries in Europe or above the age of 25, and looks at the distribution between males and females in form of a pie chart (middle).

[1]http://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic.html

HILDA'16, June 26 2016, San Francisco, CA, USA

ISBN 978-1-4503-4207-0/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2939502.2939507

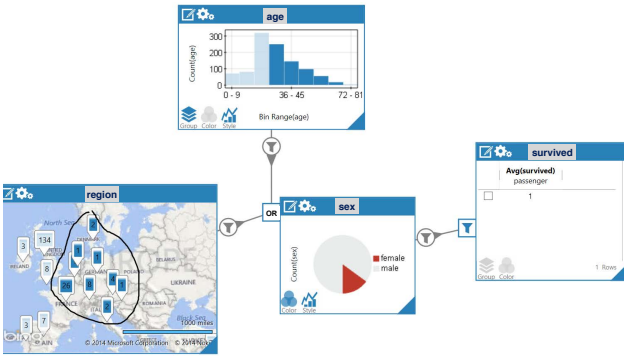


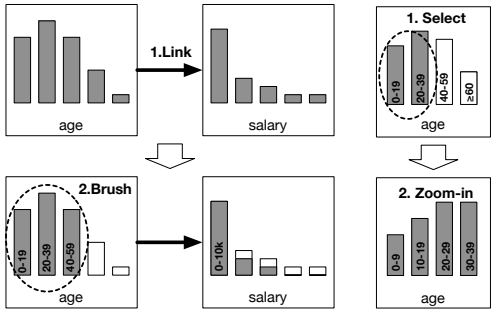
Figure 1: Interactive Data Exploration Session

Finally, he selects only the female passengers and views the average rate of survived passengers (right).

As outlined before, interactive data exploration requires visualizations that quickly react to a given user interaction (e.g., restricting the data using a filter condition). A recent study [7] has shown that visual delays of more than 500ms tend to decrease both end-user activity and data set coverage due to the reduction in rates of user interaction, which is crucial for overall observation, generalization and hypothesis. However, traditional database systems are ill-suited for speeding up visualizations. In fact, with a commercial database and the use of appropriate indexes, the aforementioned PanoramicData system still showed significant slowdowns to seconds even with small data sets of about 100MB.

Contributions: In this paper, we propose a novel multi-dimensional tree-based index, called *VisTree*, for interactive computation of histograms and related visualizations in an in-memory read-only analytical database system. In its current version, a *VisTree* can be used for a wide-range of visualizations such as histograms and their siblings, pie charts, choropleth maps, or bubble charts. These visualizations are often the default in systems like PanoramicData or Tableau [2] as they allow for a quick analysis of the value distribution (see Figure 1).

While annotated multi-dimensional indexes have already been used to compute the answer of aggregate queries efficiently [5], our work explores new techniques to better align the design of multi-dimensional indexes with the needs of visualization tools: (1) *VisTrees* are visually-balanced which means that the most commonly requested regions are pulled to the top level of our index structure, and thus allow fast computations of aggregate results over the initially visualized histogram buckets. (2) Differing from the interface of a normal database index that is designed for stateless key-



(a) Brushing-and-Linking (b) Zooming

Figure 2: Interaction Techniques

or range-queries, the *VisTree* interface is designed to support stateful bulk-oriented requests in order to natively support user interactions such as brushing-and-linking as well as zooming. (3) For data exploration, a major obstacle is the high cost required to prepare the data. In order to tackle this issue, we leverage the nature of data exploration by creating *VisTrees* incrementally on-the-fly and steered by user interactions. Moreover, in order to guarantee the interactiveness of visualizations, we support the ability to approximately query partially-loaded *VisTrees* during creation.

In our experiments, we show that *VisTrees* achieve an interactive speed below 500ms for a variety of data sets of differing characteristics with regard to data size and the number of attributes linked by the user interactions.

Outline: The remainder of this paper is organized as follows: Section 2 presents an analysis of typical user interactions on histograms and discusses requirements and design decisions for our index structure to better align database systems and visualization tools. Afterwards, Section 3 presents details of our visually-balanced *VisTree* index structure. Section 4 discusses how index creation and index searches work for *VisTrees*. Finally, Section 5 shows our initial performance results and Section 6 concludes with a summary and a discussion of future work.

2. ANALYSIS OF USER INTERACTIONS

Interactive data exploration tools primarily allow users to quickly gain insights into unknown data sets and discover interesting patterns using interactive visualizations. In the following discussion, we focus on histograms as interactive visualizations. However, the results generalize to other related visualizations such as pie charts, choropleth maps, or bubble charts.

In this section, we first discuss the basic workflow of data exploration using interactive histograms and dissect two common interaction techniques (brushing-and-linking as well as zooming) in detail. Then, we discuss requirements and design decisions for a multi-dimensional tree-based index structure to efficiently support data exploration.

2.1 Basic Workflow and User Interactions

The basic workflow of visual data exploration as known in information visualization typically follows the Visual Information Seeking Mantra: "Overview first, zoom and filter, then details-on-demand" [8]. In order to get an overview over some part of the data (e.g., a table), users often start by exploring distributions of individual attributes and then correlate attributes to detect interesting patterns and test different hypotheses. For example, as shown in the example session in Figure 1, the user first starts to analyze the distributions of individual attributes of passengers such as their origin, their age and their sex. Afterwards, the user incrementally links the attributes and applies filter conditions (female passengers above the age of 25) to analyze the effect of filters on the rate of passengers who survived.

There exist many different techniques that allow users to interact with visualizations [4]. In this paper, we focus on two important interaction techniques for histograms that are found in many exploration tools [11, 6, 10]: *brushing-and-linking* as well as *zooming*. We present these techniques next in more detail.

Brushing-and-Linking: Brushing-and-linking is an interaction technique that allows users to detect dependencies and correlations of individual attributes by using multiple "connected" visualizations. The basic idea of brushing-and-linking for histograms is shown in Figure 2(a): The user first links two histograms (upper part of Figure 2(a)), which means that he is now able to filter the attribute values of the histogram on the right hand side (*salary*) by the attribute values of the histogram on the left hand side (*age*). We call the attribute on the left hand side the *filter attribute*, and the attribute on the right hand side the *dependent attribute*. In general, brushing-and-linking can also be used with multiple filter attributes that are connected to the same dependent attribute. All filter attributes that are linked to the same dependent attribute can be connected via boolean AND/OR operators to formulate more complex filter predicates. Figure 1 shows an example, where the *sex* attribute was filtered by *region* and *age*.

Zooming: Zooming is not only used for map-based visualizations to change the resolution (by zooming-in and -out) but also for histograms to change the bucket definitions used to display the data. The basic idea of zooming in histograms is shown in Figure 2(b): The user selects a subset of buckets in one histogram and then zooms into the selected range. That way, a user can show a refined distribution for the selected value range. Zooming-out typically brings the user back to a zoom level analyzed before.

2.2 Requirements and Design Decisions

Computing a histogram can be seen as an aggregation query (i.e., count) over one dimension grouped by the different buckets (e.g., $0 \leq salary \leq 10k$, $10k < salary \leq 20k$, etc.) Moreover, if one histogram is filtered by another one (by brushing-and-linking) the aggregation query uses the second dimension as a selection predicate. For example, computing the buckets of the *salary* histogram in Figure 2(a) (lower part) can be seen as a count query on the *age* attribute, which is constrained by $0 \leq salary \leq 10k$. In the following, we discuss requirements and design decisions for our multi-dimensional index structure to support an efficient computation of histograms.

Index Structure: The goal of classical multi-dimensional tree-based indexes such as *R*-trees [1] is to keep the index balanced; i.e., they try to keep all leaves at the same depth. This achieves a good average performance for all potential region requests. Thus, the underlying assumption is that each region request is equally likely to be issued. However, for the interactive exploration workloads discussed before

this assumption does not hold.

For histograms, certain requests are more likely to occur than others because of the following reasons: First, data exploration tools often use pre-defined rules to compute the bucket definitions. For example, it is common to use the minimum and maximum value divided by the number of default buckets, or predefine the boundaries as multiples of 10 (e.g., 0-100, 101-200, 201-300, etc). Second, a similar observation holds for user interactions that can be applied to a histogram. For the brushing-and-linking interaction, the buckets used for filtering are also determined by the exploration tool. Moreover, for zooming-in, the new bucket definitions are also computed based on the selected buckets. Consequently, one design decision is that we align the upper index levels with the requests that are most likely.

Index Search: The classical lookup interfaces for multi-dimensional indexes are not designed to efficiently query histograms. They focus on a single key- or range-request (e.g., all records with a value between 0 and 100). However, histograms have a different access pattern. First, instead of returning all data points that fall into a given region, a histogram requires an aggregated result (i.e., the count) for a given region. Second, histograms require to lookup several regions at once (e.g., ages $0-19$, $20-40$, etc.) which is in contrast to the more traditional access pattern of one region at a time. Furthermore, histogram operations (e.g., zooming) are stateful operations. To that end, a histogram lookup should not always start to search the index from its root node but should be able to specify a hint from which nodes in the index to start the lookup.

Index Creation: In data exploration it is not clear a priori which attributes a user will link together. A naive approach would be to build a multi-dimensional index over all attributes of a data set. However, creating a multi-dimensional index such as an R-Tree is typically very expensive (especially for a high number of dimensions) since the data set actually needs to be sorted by all dimensions. Further, querying a high-dimensional index might lead to a poor query performance, even when visually-balancing the index.

In order to deal with this issue, we follow a different route. We only pre-create one-dimensional indexes for each attribute in the data set and user interactions steer the index creation of the multi-dimensional indexes. However, creating the full multi-dimensional index on-the-fly and then querying it once it is built can take typically much longer than the interactive threshold of 500ms for most data sets. To account for this, our index creation scheme computes approximate counts and thus allows users to execute queries on a partially loaded *VisTree*.

3. THE VISTREE INDEX

In this section, we first explain the one-dimensional *VisTree* to compute individual (non-linked) histograms. Afterwards, we present the multi-dimensional *VisTree* that is used for linked visualizations. Index creation and index searching are discussed in Section 4.

3.1 The One-dimensional VisTree

The main idea of our one-dimensional *VisTree* is shown in Figure 3. The *VisTree* is based on an annotated B^+-tree that stores count values (or other aggregate values) for the

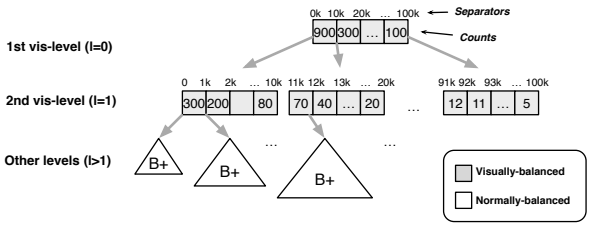


Figure 3: One-dimensional *VisTree*

different ranges represented by the separators (keys) stored inside an index node. However, as opposed to annotated indexes as presented in [5], the upper L levels of a *VisTree* are visually-balanced.

The main idea of *visually-balancing* is that the regions of an index node are aligned with histogram buckets that are typically visualized by the user interactions during data exploration as discussed in Section 2. To that end, we align the regions in the root node (level $l = 0$) with the buckets that are initially required to compute the histogram when a user first visualizes an attribute. The regions on the lower visually-balanced levels ($l > 0$ and $l < L$) represent buckets of histograms to efficiently support zooming interactions of the user. All other levels of a *VisTree* that are below level L are balanced using the normal rules of a B^+-tree.

The number of visually-balanced levels L of a *VisTree* is a tuning parameter that can be set when creating the index. Typically, L should be set to the number of zoom-levels that should be supported by the exploration tool. For example, the *VisTree* in Figure 3 has two visually-balanced levels and thus supports one additional zoom-in interaction after visualizing the initial histogram. However, if the data size is small and can be scanned in interactive time anyway, L can be also set to be smaller than the number of zoom-levels that are supported by the exploration tool.

As discussed before, the separators on the visually-balanced levels are not determined by the normal balancing rules. Instead, the separators are aligned with the histogram buckets that are requested by the data exploration tool to visualize the histograms. Therefore, when creating a *VisTree*, the separators can be specified by the exploration tool or user in two ways: In the first variant, the user can specify the separators for each level manually, which is the most flexible version; i.e., the user defines an additional table with two columns *level* and *separator*, and refers to this table when creating the index. In the second variant, the user only specifies a default number of buckets B as a hint and provides a user-defined function (UDF) that returns the list of separators for each visually-balanced level. The signature of this UDF is: `separators(buckets B, level L)->list keys`. In this paper, we use a simple UDF that implements a typical rule that is used in exploration tools such as PanoramicData [11] or Tableau [2]: for the level $l = 0$, the UDF uses the minimum and maximum value of the attribute domain that should be indexed (using meta information from the database catalog) and divides the domain range equally by the number of buckets B provided as a hint by the user. Moreover, for the other visually-balanced levels ($l > 0$ and $l < L$), the UDF further divides the sub-range into equally sized regions. For example, the *VisTree* in Figure 3 divides the domain $0 - 100k$ into $B = 10$ buckets on level $l = 0$ and then splits each bucket of the top-level into $B = 10$ further buckets on the next level $l = 1$. In the future,

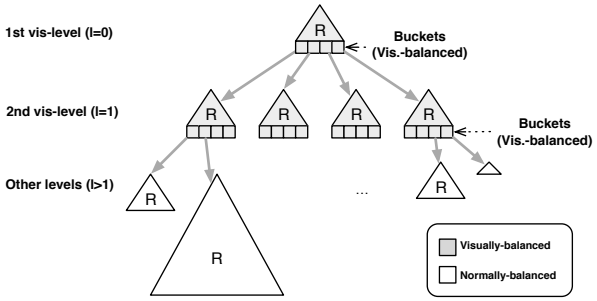


Figure 4: Multi-dimensional *VisTree*

we plan to investigate a parameter-free self-tuning scheme where the VisTree is re-balanced using statistics over the requested regions based on past user interactions.

Separating the *VisTree* into visually-balanced and normally-balanced levels can lead to an annotated tree that is imbalanced in total (especially for skewed data). However, as discussed before, requests for the (upper) visually-balanced levels are much more likely. Moreover, users start data exploration on the upper visual levels and iteratively zoom into the histogram to the next levels. If zooming needs to read the normally-balanced levels, we provide an approximate query processing scheme discussed next in Section 4. This enables interactive performance below 500ms independent of both the data size and the number of visually-balanced levels L that are used.

3.2 The Multi-dimensional VisTree

The index structure of our multi-dimensional *VisTree* is shown in Figure 4. The multi-dimensional *VisTree* is used to support brushing-and-linking interactions where the user filters on one or more attributes. In general, the multi-dimensional *VisTree* extends the concepts discussed previously through the following two ideas.

Firstly, instead of a B^+-tree, we use an annotated R-tree [1] as basis. Each index node stores multi-dimensional regions (rather than one-dimensional ranges) and their annotated aggregate values. Similar to a one-dimensional *VisTree*, the upper L index levels are visually-balanced, the other levels are normally-balanced following the rules of an R-tree. As for the one-dimensional *VisTree*, the separators are provided using one of the variants discussed previously.

Secondly, for each visually-aligned level, we additionally build internal R-tree nodes if the number of regions within a node is higher than the fill-grade of a normal R-tree node. These additional R-tree nodes on top of a visually-balanced node support efficient filter operations (for brushing-and-linking). This is necessary, as the number of regions that need to be stored per visually-balanced node grows exponentially with the dimensions and is no longer fixed as it was in the one-dimensional case. More precisely, for D dimensions and B buckets per dimension we need to store B^D regions in a visually-balanced node.

4. INDEX CREATION AND SEARCHING

In this section, we discuss how index creation and index searches work for *VisTrees*.

4.1 On-the-Fly and User-Steered Index Creation

Indexing data for data exploration is a challenging task

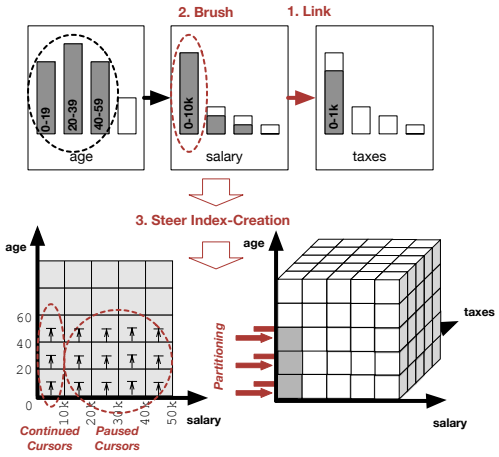


Figure 5: Steering the Index Creation

since it is unclear a priori which attributes the user will link together and, hence, which multi-dimensional *VisTrees* to create. Moreover, creating one single high-dimensional *VisTree* over all attributes of a data set is seen as not desirable due to the high index creation cost and the poor query performance of multi-dimensional indexes (even when visually-balancing the tree, as shown in our experiments in Section 5). However, an important observation is that users typically link only a smaller number of attributes when exploring data sets with a tool such as PanoramicData [11]. To that end, we follow a different route in this paper as described in the following.

On-the-Fly Index Creation: Rather than pre-creating a single high-dimensional *VisTree* over all attributes, we only pre-create one-dimensional *VisTrees* (i.e., one for each attribute) and create multi-dimensional *VisTrees* on-the-fly when users drag and link attributes on the screen. The reason is that the user typically drags out the attribute without immediately applying a follow-up interaction such as brushing-and-linking or zooming. Therefore, we leverage this interaction pause from the user to create the high-dimensional *VisTrees* in the meantime.

In general, *VisTrees* can also be created eagerly (i.e., when a database system is idle) without any user interaction triggering the index creation process. Furthermore, high-dimensional *VisTrees* that are not used anymore in a user session can be garbage collected. Both garbage collection and predicting which indexes to create eagerly are not in the scope of this paper and present interesting avenues of future work.

User-steering: A second technique to minimize the index creation time is to load only those parts of the data into the high-dimensional *VisTree* that are required for a user interaction. The reason is that a *full* high-dimensional *VisTree* on the selected attributes is typically not required to display the results of a user interaction.

In our implementation, we therefore create an n-dimensional VisTree top-down in an incremental way. This takes advantage of most users' interactive pattern of first dragging out an attribute and then zooming in. That way, steering reduces the required index size. To compute the histogram for an interaction, only the first visually-balanced level that satifisfies the current zoom level is required. To make the creation process steerable for brushing-and-linking, we keep a separate cursor for each region in the $n-1$ dimensional

index selected by the current filtering predicates. An additional filter is added once the user performs a brushing interaction. This prioritizes the partitioning of selected regions, while pausing the cursors of unselected regions.

Figure 5 shows an example for steering index creation for brushing-and-linking. In this example, the user links the *tax* histogram to the *salary* histogram and applies a filter to visualize the *tax* distribution for low income ranges. As a result of steering, the creation process continues to only read data from those cursors of the 2-dimensional *VisTree* that qualify for the newly added predicate and pauses from reading data from all other cursors.

4.2 Approximate and Exact Index Searches

Although user-steered index creation only loads those parts of a *VisTree* necessary for a particular user interaction, the time needed can still exceed the interactive threshold of 500ms, especially if the data set is large or the number of dimensions is high. As our experiments in Section 5 will show, creating the first level of an 4 dimensional *VisTree* for 60M tuples can already take significantly longer than 1s. In order to support interactive computation of histograms while the index is still being created, we support searching a partially loaded *VisTree* by leveraging a variant of online aggregation (OA) [3] (called *VisTree*-OA) to compute aggregate queries.

Approximate Index Searches: The basic idea of *VisTree*-OA is that we piggybacked an online aggregation operator on the index creation process described before; i.e., the top-down index process additionally computes an approximate frequency count for each region of an index node that is being loaded. Once the user executes an interaction that requires an update of a histogram (i.e., by applying a filter after linking), the *VisTree*-OA consumes these approximate counts to compute the requested histogram buckets and returns an error interval for each bar. As opposed to the traditional online aggregation [3], *VisTree*-OA does not need to scan the complete base table, but instead only needs to scan the relevant regions from the $n-1$ dimensional *VisTree*. Moreover, *VisTree*-OA also knows the (exact) count values of the regions of the $n-1$ dimensional index (if this index is already fully loaded) and thus can correct the count estimates if they are overestimated (e.g., due to data skew or data ordering caused by scanning the data of the n-th dimension). Hence, as we will show in our evaluation in Section 5, *VisTree*-OA typically computes tighter error bounds in the same amount of time when compared to traditional online aggregation especially for low selectivities; i.e., if the user has selected only a few bars in the linked histograms.

Exact Index Searches: Once a *VisTree* is fully loaded, it can be used by exact index searches that leverage the visually-balanced levels as described in Section 3. In order to optimize exact index searches for computing histograms interactively, we implement two techniques for our index search interface: (1) An index search can request all buckets at a time to compute the required histogram (i.e., *VisTrees* offer a bulk interface). (2) Index searches are stateful; i.e., the user can specify a hint on which level of the index the search should start. That way, zooming-in interactions do not need to scan the complete index. For zooming-out, the index interface does not provide any extensions since this can be efficiently supported by a query cache available in almost any database system.

5. EXPERIMENTAL EVALUATION

In this section, we report the results of our initial experimental evaluation using *VisTrees*. We analyze the performance of index creation and the two querying schemes as presented in Section 4. As the data set for all experiments, we generated a synthetic table with 60M tuples and 8 attributes (using a correlation factor of 0.5 for all attributes). Moreover, for all *VisTree* indexes in this evaluation, we used $L = 3$ (i.e., three visually-aligned levels) and $B = 10$ (i.e., 10 buckets per node and dimension) to compute the separators as outlined in Section 3 using equi-width buckets for the visually-aligned levels.

We implemented our *VisTree* index structure in C++ 11 and compiled our code using GCC 4.9.2. For executing the experiments, we used a machine with an Intel Xeon E5-2660 v2 processor (10 cores) and 256GB RAM running the Ubuntu 14.01 Server Edition as operating system. Furthermore, for all experiments in this paper we used only a single thread for the execution of the index creation and index search to show the direct effect of all our optimizations.

5.1 Exp. 1: Index Creation

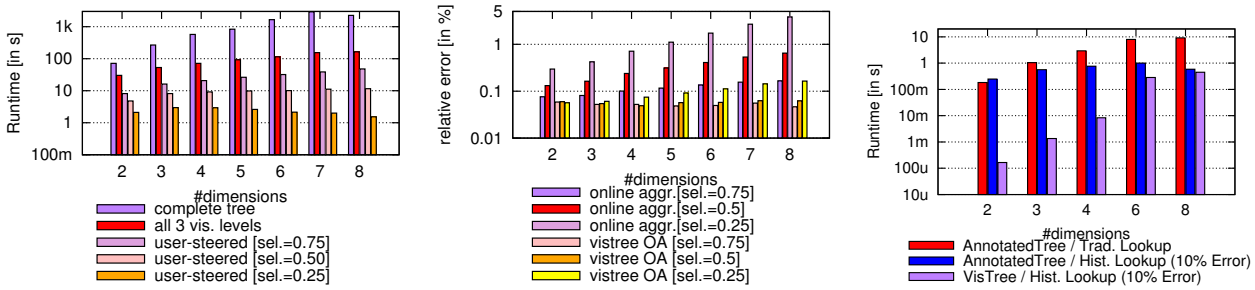
In our first experiment, we analyzed the index creation time when applying user interactions to steer the creation. To that end, we created an n-dimensional *VisTree* with three visually-balanced levels using an existing $n-1$-dimensional *VisTree* for user-steering. In the experiments, we varied n from 2 to 8. Moreover, for each of the $n-1$-dimensions, we applied a filter predicate with a given selectivity ranging from low (0.25) to high (0.75) selectivity to simulate a steering interaction. In all experiments, the steering interaction was executed directly after starting the loading process without any user interaction pause.

To show the effect of user-steering, we only load the required parts of the first visually-balanced level of a *VisTree*. In order to study the savings of user-steering, we executed two other baselines: (1) the time to load the complete *VisTree* and (2) the time to load all three visually-balanced levels without applying the filter predicates for steering.

Figure 6(a) shows the indexing time to create the *VisTree* with user-steering (*user-steered [sel.=...]*) and without (*complete tree* and *all 3 vis. levels*). While the indexing time with user-steering is in the order of a few seconds, loading the full *VisTree* typically is in the order of minutes for 8 dimensions. Moreover, another interesting aspect is the effect of the selectivity for the user-steered index creation. While the indexing time for a selectivity of 0.75 slightly increases with increasing dimensions, the indexing time for a selectivity of 0.25 decreases. This is due to the low selectivity; reducing the amount of data while increasing the number of dimensions outweighs the higher cost to create the VisTree.

5.2 Exp. 2: Approximate Searches

As discussed in Section 4.2, *VisTrees* can already be queried approximately by using our *VisTree* online aggregation (*VisTree*-OA) during index creation. This technique is required to meet the interactivity threshold of 500ms since even with user-steered loading, the index creation might take longer. For the index search, we query the first visually-balanced level and apply the same filter predicates that we used in the experiments before. Moreover, we start the index search directly when the index creation starts. This simulates a user who links a new attribute and immediately



(a) Indexing Time (b) Approx. Search (c) Exact Search

Figure 6: Index Creation and Search: Varying # Dimensions

applies a filter without any *think time* in between.

Figure 6(b) reports the error achieved at 500ms of *VisTree*-OA and compares it to the traditional online aggregation (OA) [3] as a baseline. The results show that *VisTree*-OA converges to much smaller errors when compared to OA, especially at lower selectivities. The reason is that *VisTree*-OA only needs to read the relevant data using the existing $n-1$ dimensional index, whereas OA needs to scan the table. Secondly, *VisTree*-OA can leverage counts on linked attributes to obtain tighter conservative bounds on the estimate, while OA assumes the cardinality of the table.

5.3 Exp. 3: Exact Searches

In this experiment, we show the runtime of exact index searches for an n-dimensional *VisTree* once it is fully created. As a workload, we executed a histogram query which applied a filter on $n-1$ dimensions using a selectivity of 0.5 for each filter attribute. Furthermore, in this experiment, the requested buckets matched the topmost visually-balanced level $l = 0$. For computing a histogram with n dimensions, we used a *VisTree* index that exactly contained only the required attributes for all n dimensions.

Figure 6(c) shows the results of this experiment. As a baseline, we compared the runtime to an annotated R-tree as presented in [5]. For the annotated R-tree, we implemented two index search strategies: *Trad.-Lookup*, which executes one request per bucket, and our *Hist.-Lookup*, which uses the same index search as the *VisTree* index (bulk+stateful). We see that the *VisTree* clearly outperforms the annotated R-tree when using the trad. lookup interface, which has a runtime of more than 500ms for more than 3 dimensions. When using the Hist.-Lookup interface, the perfomance of the annotated R-tree is improved, but it is still not as good as our *VisTree* index. It is only upon reaching 8 dimensions that the performance of both indexes is similar, and even then the *VisTree*'s performance is slightly better. This is because the number of regions on the topmost visually-aligned level already contains 10^8 regions (since we are using 10 buckets per dimension). Since our data set contains only 60M records in total, the first level of our *VisTree* (including its internal R-tree nodes) has a similar height as the complete annotated R-tree.

6. CONCLUSION & FUTURE WORK

In this work, we presented a novel multi-dimensional tree-based index structure called *VisTree* that efficiently supports the interactive computation of histograms and their siblings (pie charts, choropleth maps, or bubble charts). As key contributions, this paper explored techniques to better align the design of multi-dimensional indexes with the needs of visualization systems for data exploration. In our experiments, we showed that the *VisTree* achieves a speed-up of up to three orders of magnitude compared to traditional multi-dimensional annotated indexes and thus enables an interactive speed of below 500ms for computing histograms, even on large data sets and multiple dimensions.

As an avenue of future work, we plan to look into other visualizations (e.g., time series). Moreover, other topics such as garbage collection and the prediction of which indexes to eagerly create are interesting avenues of future work.

7. ACKNOWLEDGEMENTS

This research is funded partly by the Intel Science and Technology Center for Big Data, the NSF CAREER Award IIS1453171, the Air Force YIP AWARD FA95501510144, NSF IIS1514491, and gifts from SAP, Oracle, Google, and Mellanox.

8. REFERENCES

[1] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, 1984.

[2] P. Hanrahan. Analytic database technologies for a new kind of user: The data enthusiast. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, 2012.

[3] J. M. Hellerstein et al. Online Aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, 1997.

[4] D. A. Keim. Information visualization and visual data mining. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1), 2002.

[5] I. Lazaridis et al. Progressive Approximate Aggregate Queries with a Multi-resolution Tree Structure. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, 2001.

[6] Z. Liu et al. imMens: Real-time Visual Querying of Big Data. *Computer Graphics Forum*, 32(3pt4), 2013.

[7] Z. Liu et al. The Effects of Interactive Latency on Exploratory Visual Analysis. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12), 2014.

[8] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, 1996.

[9] C. Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional relational databases. *Visualization and Computer Graphics, IEEE Transactions on*, 8(1), 2002.

[10] P. Terlecki et al. On Improving User Response Times in Tableau. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[11] E. Zgraggen et al. PanoramicData: Data Analysis through Pen and Touch. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12), 2014.