Eurographics Symposium on Parallel Graphics and Visualization (2022)
R. Bujack, J. Tierny, F. Sadlo (Editors)

Rainbow: A Rendering-Aware Index for High-Quality Spatial Scatterplots with Result-Size Budgets

Qiushi Bai, Sadeem Alsudais, Chen Li, and Shuang Zhao

University of California, Irvine, CA 92697, USA

Abstract

We study the problem of computing a spatial scatterplot on a large dataset for arbitrary zooming/panning queries. We introduce a general framework called "Rainbow" that generates a high-quality scatterplot for a given result-size budget. Rainbow augments a spatial index with judiciously selected representative points offline. To answer a query, Rainbow traverses the index top-down and selects representative points with a good quality until the result-size budget is reached. We experimentally demonstrate the effectiveness of Rainbow.

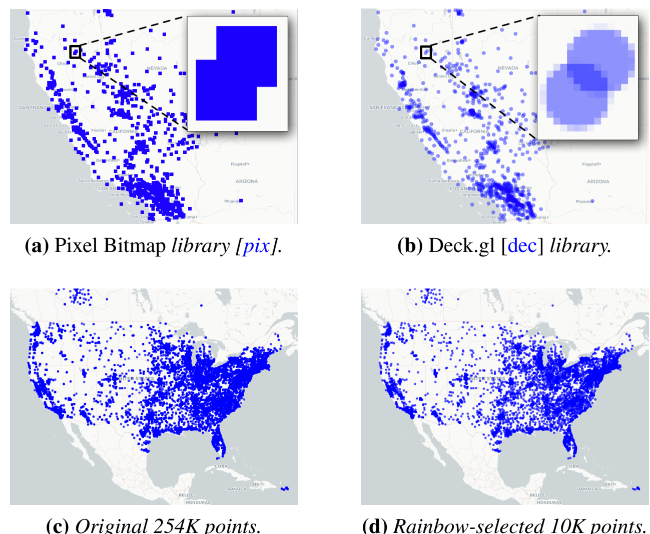
CCS Concepts

• ***Information systems*** $\rightarrow$ ***Middleware for databases;*** • ***Human-centered computing*** $\rightarrow$ ***Visualization systems and tools;***

1. Introduction

Scatterplot is an important visualization method to explore spatial data. To support arbitrary zooming, panning, and clicking operations, a typical visualization system adopts a client-server architecture. A frontend library on the client translates each user operation to a query to the server and renders the points returned from the server. Every query should be served efficiently, preferably in milliseconds [LH14]. However, when the data is large, computing, transferring, and rendering the result points can be expensive, especially when the client is using a mobile device with limited resources and connected to a slow network. One idea is to reduce the number of points transferred and rendered through sampling or aggregation. A natural requirement is that the rendering of the sample should be as similar as possible to the rendering of the original result. The client using different frontend libraries such as Mapbox [map], Leaflet [lea], or Deck.gl [dec] may have different rendering effects for the same set of spatial points, as shown in Figure 1 (a) and (b). This difference can be significant when different rendering parameters are applied, such as the colors and sizes of the points. Thus, the diversity of frontend libraries is a challenge for developing general-purpose techniques that compute high-quality scatterplots.

In this paper, we study how to efficiently compute a set of spatial points from a dataset for an arbitrary spatial range query with a result-size budget, e.g., sending up to 10,000 points to the client. We introduce a general framework called "Rainbow," which stands for "Rendering-Aware Index for Big Scatterplot Workload." Its main idea is to augment a spatial index with judiciously selected representative points to maximize the quality of the rendered result. To answer a spatial query from the client with a result-size budget, Rainbow traverses the spatial index top-down and greedily selects representative points with a maximal quality of the rendered result until the number of points reaches the budget (Figures 1(c) and (d)). This technique can not only generate a high-quality scatterplot within the budget, but also be applied with any user-specified rendering function and quality metric.



(a) Pixel Bitmap *library [pix].* **(b)** Deck.gl [dec] *library.*
(c) *Original 254K points.* **(d)** *Rainbow-selected 10K points.*

Figure 1: *(a)* Pixel Bitmap *maps a point to a square of solid pixels. (b)* Deck.gl *maps a point to a circle of pixels with opacity. (c) and (d) are rendered by the same Deck.gl on (c) the original dataset of 254K points and (d) the 10K points selected by Rainbow.*

2. Related Work

There are many studies of spatial visualization, and we focus on techniques that support efficient zooming and panning operations.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

DOI: 10.2312/pgv.20221060

Pre-aggregation-based approaches pre-compute image tiles or datacubes offline and then answer zooming/panning queries online efficiently. (1) *Image-tiling* techniques [EMJ16, YZS18, CSK*13] pre-compute image tiles and return pre-rendered images as query results. For example, AID [GEJ19] and AID* [GE20] minimize the index size and construction time by considering the costs of processing and storing image tiles and data tiles. (2) *Datacube* techniques [LKS13, MCW*20, LWSY20, TdLPRC18] store a density value (i.e., a count of objects) on each pre-defined cell and return cell centers along with their density values as query results. These approaches do not support frontend libraries (e.g., the scatterplot layer in Deck.gl) that take the original points' accurate coordinates as input to render scatterplots. Also they do not allow an arbitrary result-size budget provided by the client.

Sampling-based approaches [MFDW17, FPDmcs12, JS20, GFCB18] return a sample of the original points for online zooming/panning queries. (1) *Offline-sampling* approaches (e.g., VAS [PCM16], Sample+Seek [DHC*16]) draw a sample on the raw data offline, and run online queries on the sample instead of on the raw data to achieve a high efficiency. (2) *Online-sampling* approaches (e.g., RS-Tree [WCLY15] and Pyramid [CZF*22]) build a hierarchical index offline with sampled points on nodes and then return a sample of points for an online query. For (3) *3D-point clouds*, similar ideas have been proposed, e.g., [Sch14, Sch21, SOW20]. The sampling strategies of these approaches are tailored to some specific rendering effects and quality measures, and they cannot adapt to new rendering or quality functions. For example, as shown in [PCM16], random and stratified sampling strategies do not provide high-quality scatterplots for regression-analysis tasks. The proposed framework can adapt to desired sampling schemes driven by user-defined rendering and quality functions.

Other approaches include progressive visualization techniques [MFDW17, CGZ*16, JLZ*16], prefetching-based techniques [BCS16, YMS17, TLD*19], special indexes [EZBK16] designed for histograms, systems [PSF12] designed specifically for earth data, and query acceleration techniques [WLX19] specialized for encrypted data. These techniques do not adapt to different frontend libraries given a result-size budget while our approach does.

In summary, the proposed framework is unique in three folds: 1) it supports arbitrary frontend libraries that require actual point objects as input and render results on the fly; 2) it makes the best effort to compute a high-quality sample given an arbitrary result-size budget; 3) it adapts to different rendering and quality functions by considering them in its offline index-building and online query-processing phases.

3. Problem Formulation

We consider a client-server architecture consisting of a server and multiple clients. Given a dataset D on the server, a rendering function R, and an error function Δ, the server loads D and builds an in-memory data structure to answer arbitrary visualization queries submitted by the clients. A client query is in the format of $Q(r,x,b)$, where r is a spatial range, x is a resolution, and b is a result-size budget. In this paper, we focus on the problem of finding a set of points S such that $S \subseteq D_r \subseteq D$ and $|S| \leq b$, where $D_r = \{p : p \in D \cap r\}$, and $\Delta(R(D_r), R(S))$ is minimized. In other words, the server selects a subset of points to minimize the error between the rendered result of the subset and the rendered result of the original set of points in the query range.

Rendering functions: There are various ways to render a spatial point as a set of pixels on the client screen. For example, Deck.gl [dec] utilizes WebGL to apply a series of geometric transformations and projections, and then rasterizes a point on a set of colored pixels (as shown in Figure 1(b)). The complex graphics pipeline is not a focus of this paper. We can conceptually model it as a rendering function that maps a spatial point to a set of colored pixels, where the opacity value of each pixel can be decided by the distance of the pixel to the projected location of the point. For instance, as shown in Figure 1, to obtain a similar result, the rendering function in (b) requires more points in dense areas than the function in (a) because the opaque pixels in (b) are harder to be over-plotted.

Error functions: To measure the quality of S, we compare the image rendered from S with the image rendered from the original result set D_r. Existing image similarity metrics such as MSE [PSC00] and SSIM [WBS*04] can be adopted. In applications that are more concerned with geometric distances between spatial objects such as clustering analysis, distance metrics comparing two sets of spatial objects can also be used, such as Within-Cluster Sum of Squares (WCSS) [DV*15]. Note that image similarities do not directly represent visualization faithfulness, and our framework can be adopted if a faithfulness-oriented quality function is given.

4. Rainbow

Rainbow is a framework that achieves interactive visualization using an offline index-construction phase and an online tree-traversal phase. During the offline phase, we build a spatial index on dataset D. The index is a tree-based structure (e.g., K-D tree or quadtree) that recursively partitions the spatial data into disjoint smaller regions. We then apply the given rendering function R and the error function Δ to augment each internal node with representative points. The goal is to minimize the error between the result of rendering the representative points and the result of rendering all data points in the subtree under the node. We also maintain the error information on each node to be used in the online phase.

To answer a visualization query $Q(r,x,b)$, we traverse the tree top-down with an initial result set consisting of only the representative points at the root node. We expand the result set iteratively until either the size budget is reached, or the error between the rendering effect of the result set and the raw dataset within the query range is small enough. A feasible result set is modeled as a tree cut that consists of a set of disjoint nodes spanning the query range. We iteratively expand the tree cut in a greedy manner to find a result set that maximizes the visualization quality. We use quadtree as an example spatial index in the following discussion.

4.1. Offline Index-Construction Phase

Rainbow extends the typical quadtree by adding representative points on each node so that the error of rendering these representative points for the region compared to rendering all points under the subtree is minimized. For simplicity, we consider the case of selecting a single representative point per node in the following discussion, and it can be generalized to select multiple points per node.

We first build a quadtree on the entire dataset D, then traverse the tree bottom-up to select a representative point for each node. As shown in Figure 2, suppose Ω_N is the spatial region covered by node N, and D_N is the set of points in D located inside Ω_N, namely

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

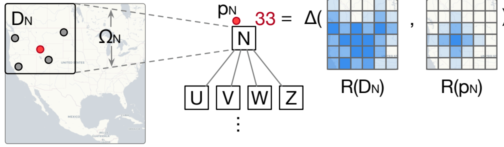


Figure 2: *Rainbow selects a representative point on each tree node and stores the error.*

$D_N = \{p : p \in D \cap \Omega_N\}$. We select a point p_N from D_N such that

$$\forall p_i \in D_N : \Delta\big(R(D_N), R(p_N)\big) \leq \Delta\big(R(D_N), R(p_i)\big).$$

In other words, the error of rendering p_N against rendering all points under node N is minimized compared to rendering any other point p_i in D_N. We call p_N a *representative point* in the node N. We also store this error $\Delta(R(D_N), R(p_N))$ in node N, which will be used in the online tree-traversal phase as explained in Section 4.2. Computing $R(D_N)$ needs to render all points in the subtree, which can be costly. To solve the problem, we can approximate $R(D_N)$ by only rendering the representative points in N's four children. We can also reduce the candidate set size by selecting p_N only from the representative points in N's four children.

Different resolutions: A rendering function takes not only the point to be rendered as input, but also the resolution of the output image as a parameter. Different visualization queries can have different zoom levels that correspond to different resolutions. Given a specific query range r and a resolution x, a node N in the tree represents a part of the final output image. N also has its own resolution decided by r and x. If we only select the representative point for a specific query resolution, it might not have the minimum error for other resolutions. There are two ways to address this problem. One is that on each node, for each zoom level, we select the best representative point and its error. This approach is feasible when the total number of zoom levels is small (e.g., Mapbox supports at most 23 levels). Another strategy is to select a single representative point for all the zoom levels to reduce the storage overhead, e.g., by using the most common point.

4.2. Online Tree-Traversal Phase

Given a query $Q(r, x, b)$, we traverse the tree top-down and expand the result set iteratively until either $\Delta(R(D_r), R(S)) = 0$ or $|S| = b$. Formally, let the index tree be T. A feasible result set is modeled as a tree cut κ where $\kappa = \{n_i : n_i \in T \wedge n_i \cap r \neq \emptyset \wedge \forall j \neq i, n_i \cap n_j = \emptyset\}$. In other words, a tree cut is a set of disjoint nodes from the tree that overlap with the query range. A result set S is the union of all representative points stored on each node in the tree cut κ.

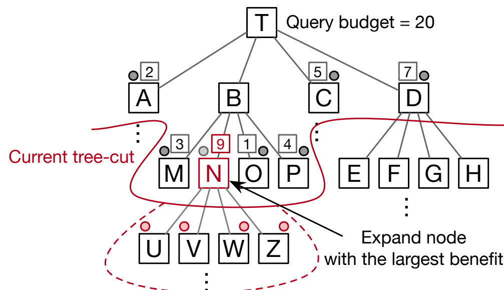


Figure 3: *In the tree-traversal phase, Rainbow iteratively expands the tree cut by selecting the node with the largest benefit value and replacing it with its four children, until the size budget is reached.*

Best-first search: We do best-first search to expand the tree cut. For each node in the current tree cut, we compute a benefit value. In each iteration, we select the node with the largest benefit value and replace it with its four child nodes. The replacement will trigger the computation of the benefit values for the four child nodes, but will not affect all other existing nodes in the tree cut. We repeat this process until we get enough result points ($|S| = b$), or all the nodes in the tree cut have a zero benefit value, which means further expanding the tree cut will not increase the quality of the visualization. Figure 3 shows an example. Suppose the current tree cut κ has nodes $\{A, M, N, O, P, C, D\}$ and the result set S has 7 points, smaller than the budget 20. The benefit of each node is listed over its box. We choose node N with the largest benefit among the nodes in κ to expand it to its four children U, V, W, and Z.

Computing benefit: The benefit value guides the search to expand the node that reduces the error the most while adding the least number of points into the result set. We define a gain value that indicates if a node N is expanded, how much error can be reduced between the current result set and the original result set. Suppose $E_N[x]$ is the error stored in node N, which is computed by comparing the rendered result of the representative point p_N against the rendered result of the points in the subtree D_N using the query resolution x in the offline building phase, namely $E_N[x] = \Delta(R(D_N), R(p_N))$. Let U, V, W, and Z be the four child nodes of N. Then the errors of rendering the representative point against the subtree using the query resolution x on each node are $E_U[x]$, $E_V[x]$, $E_W[x]$, and $E_Z[x]$, respectively. The gain value of expanding node N is computed as:

$$gain_N = E_N(x) - (E_U[x] + E_V[x] + E_W[x] + E_Z[x]).$$

Then we define a cost value that reflects how many more points will be added to the result set if we expand node N. Let C_N be the number of representative points on node N, and C_U, C_V, C_W, and C_Z be the number of representative points in the four children of N, respectively. The cost value of expanding node N is computed as:

$$cost_N = (C_U + C_V + C_W + C_Z) - C_N.$$

We formally define the benefit value of expanding node N as

$$benefit_N = gain_N / cost_N.$$

Since each node has all the information required in the definition, the overhead to compute the benefit values is small, making the online tree-traversal phase efficient.

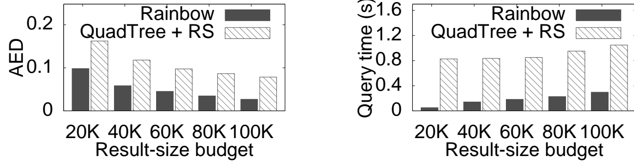
5. Experiments

We evaluated Rainbow on 100 million (17GB) geo-located tweets and compared it with three related approaches. (1) *QuadTree + RS*. It indexed the dataset using a quadtree and stored points on tree nodes. Given a range query with a size budget, it retrieved all points in the range and returned a random sample (RS) within the budget size; (2) *Nanocubes* [LKS13]. It indexed the dataset using a quadtree and stored count of points on tree nodes. Given a query, it returned the centroids of all tree nodes in the query range. Since Nanocubes did not support a result-size budget, we used the result size from Nanocubes as the budget to query Rainbow such that they had the same result size. (3) *AID** [GE20]. It is a hybrid index with both image and data tiles embedded in an R-tree. We implemented Rainbow in Java 8. All experiments ran on a Ubuntu 14 server with 4 Intel Xeon X5670 CPUs, 96GB RAM, and a 2-TB HDD drive.

5.1. Visualization Quality

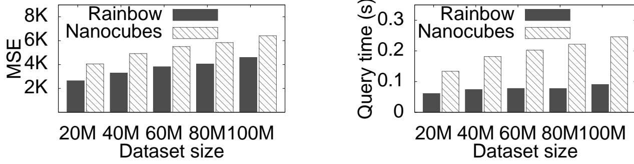
We used two rendering functions. The first one (denoted as "R1") mapped each point to a solid-colored pixel on the screen. We implemented R1 using WebGL. The other (denoted as "R2") mapped

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.



(a) *Quality (lower AED is better).* **(b)** *Query time.*

Figure 4: *Quality and query time of Rainbow and Quadtree + RS for different budget sizes using R1 and AED on 100M tweets.*



(a) *Quality (lower MSE is better).* **(b)** *Query time.*

Figure 5: *Quality and query time of Rainbow and Nanocubes for the same budget sizes (35K, 43K, 48K, 52K, and 57K, respectively) using R2 and MSE on different dataset sizes.*

each point to a set of colored pixels, and each pixel's opacity was decided by its distance to the point. We used the ScatterplotLayer API of Deck.gl as R2. We used two error functions to measure the quality of the rendered results. One was Average Euclidean Distance (AED), a variation of WCSS [DV*15]. For each point in the original query result set, we computed the distance between the point and its nearest neighbor point in the rendered result of the approach under evaluation. The average distance of all points in the original result set was the AED value of the rendered result. The other was Mean Squared Error (MSE) [PSC00], which computed an average squared pixel-wise density difference. We used AED to measure the quality of results rendered by R1 and MSE to measure the quality of results rendered by R2.

R1+AED: We varied the budget size from 20K to 100K for the 100M tweet dataset and compared the quality of the results using the same query, which was a scatterplot of the whole US. As shown in Figure 4(a), Rainbow had a better quality compared to Quadtree + RS for the R1 rendering function and AED error function.

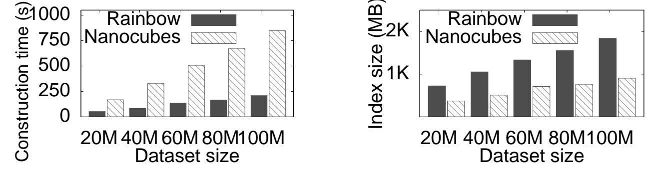
R2+MSE: We varied the raw dataset size from 20M to 100M. As shown in Figure 5(a), Rainbow achieved a higher quality than Nanocubes using the R2 rendering function and MSE error function. The reason was that Nanocubes used aggregated centroids as the result points without considering the effects of the rendering function.

5.2. Query Performance

To measure the query performance, we used the same aforementioned R1+AED and R2+MSE combinations. For the R1+AED configuration, we varied the budget size from 20K to 100K on the 100M tweet dataset, and the results are shown in Figure 4(b). For the R2+MSE configuration, we varied the raw dataset size from 20M to 100M, and the results are shown in Figure 5(b). In these configurations, Rainbow outperformed both baseline approaches, and it was able to provide an interactive query performance ($\leq 500ms$) for 100 million tweets on a single machine.

5.3. Construction Time and Index Sizes

We used the R1+AED configuration to evaluate the construction time and index size of Rainbow and Nanocubes. We varied the dataset size from 20M to 100M. As shown in Figure 6, the construction of Rainbow was faster than Nanocubes, while Nanocubes was more space-efficient. The reason was that Rainbow stored additional information on intermediate nodes on the index.



(a) *Index construction Time.* **(b)** *Index Size.*

Figure 6: *Offline construction time and index size of Rainbow and Nanocubes on different dataset sizes.*

5.4. Comparison with AID*

We constructed indexes offline on the 100M tweet dataset using Rainbow and AID*. The performance of AID* depends on the portion of tiles stored as images or data files. We showed the results of two extreme cases using AID*: one case (denoted as AID_i^*) stored all tiles as pre-rendered images and the other case (denoted as AID_d^*) stored all tiles as raw data files. It took Rainbow 209 seconds to construct the index of size $1,840$MB. In comparison, it took AID_i^* and AID_d^* $1,025$ seconds and 756 seconds to build their indexes of sizes $7,520$MB and $7,169$MB, respectively. We used five queries with different zoom levels to evaluate all three approaches. As shown in Figure 7, Rainbow had comparable query time with AID_i^*. Both outperformed AID_d^* significantly because AID_d^* had to scan the data in the query range to render the images on-the-fly.

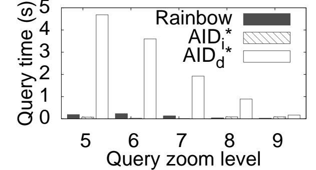


Figure 7: *Size-budget=100K.*

Table 1: *Query time and result quality of Rainbow and AID*.*

Query zoom level	5	6	7	8	9
Rainbow query time (s)	0.190	0.233	0.134	**0.043**	**0.027**
AID_i^* query time (s)	**0.074**	**0.026**	**0.004**	0.088	0.090
AID_d^* query time (s)	4.684	3.598	1.923	0.890	0.167
Rainbow result MSE (K)	1.693	1.249	0.168	0.0	0.0

As shown in Table 1, Rainbow outperformed AID_i^* for zoom levels 8 and 9. Both provided interactive query time ($\leq 500ms$) for all zoom levels. AID_i^* loaded the pre-computed images in parallel, and Rainbow could be further optimized using multi-threads. The last row shows the MSE of the approximate results (size-budget = 100K) returned by Rainbow. The MSE of the results from AID_i^* and AID_d^* were all zero since they did not do approximation.

6. Conclusion

In this paper, we developed a general framework called Rainbow that generates a high-quality spatial scatterplot for a given result-size budget. It augments a spatial index with judiciously selected representative points offline, and traverses the index to select a high-quality subset of points to answer queries online. Our experiments demonstrated the efficacy and efficiency of Rainbow. Rainbow is open-sourced at https://github.com/ISG-ICS/rainbow.

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.

References

[BCS16] BATTLE L., CHANG R., STONEBRAKER M.: Dynamic prefetching of data tiles for interactive visualization. In *Proceedings of SIGMOD* (2016), ACM, pp. 1363–1375. 2

[CGZ*16] CROTTY A., GALAKATOS A., ZGRAGGEN E., BINNIG C., KRASKA T.: The case for interactive data exploration accelerators (ideas). In *Proceedings of HILDA@SIGMOD* (2016), ACM, p. 11. 2

[CSK*13] CHENG D., SCHRETLEN P., KRONENFELD N., BOZOWSKY N., WRIGHT W.: Tile based visual analytics for twitter big data exploratory analysis. In *IEEE Big Data* (2013), pp. 2–4. 2

[CZF*22] CHEN X., ZHANG J., FU C., FEKETE J., WANG Y.: Pyramid-based scatterplots sampling for progressive and streaming data visualization. *IEEE Trans. Vis. Comput. Graph. 28*, 1 (2022), 593–603. 2

[dec] Deck.gl. https://deck.gl/. 1, 2

[DHC*16] DING B., HUANG S., CHAUDHURI S., CHAKRABARTI K., WANG C.: Sample + seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), pp. 679–694. 2

[DV*15] DUONG K.-C., VRAIN C., ET AL.: Constrained minimum sum of squares clustering by constraint programming. In *International Conference on Principles and Practice of Constraint Programming* (2015), Springer, pp. 557–573. 2, 4

[EMJ16] ELDAWY A., MOKBEL M. F., JONATHAN C.: Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (2016), pp. 601–612. 2

[EZBK16] EL-HINDI M., ZHAO Z., BINNIG C., KRASKA T.: Vistrees: fast indexes for interactive data exploration. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016* (2016), Binnig C., Fekete A. D., Nandi A., (Eds.), ACM, p. 5. 2

[FPDmcs12] FISHER D., POPOV I. O., DRUCKER S. M., M. C. SCHRAEFEL: Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *CHI Conference on Human Factors in Computing Systems, CHI '12, Austin, TX, USA - May 05 - 10, 2012* (2012), pp. 1673–1682. 2

[GE20] GHOSH S., ELDWAY A.: Aid*: A spatial index for visual exploration of geo-spatial data. *IEEE Transactions on Knowledge and Data Engineering* (2020), 1–1. 2, 3

[GEJ19] GHOSH S., ELDAWY A., JAIS S.: AID: an adaptive image data index for interactive multilevel visualization. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019* (2019), IEEE, pp. 1594–1597. 2

[GFCB18] GUO T., FENG K., CONG G., BAO Z.: Efficient selection of geospatial data on maps for interactive and visualized exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018* (2018), pp. 567–582. 2

[JLZ*16] JIA J., LI C., ZHANG X., LI C., CAREY M. J., SU S.: Towards interactive analytics and visualization on one billion tweets. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016* (2016), pp. 85:1–85:4. 2

[JS20] JIA YU, SARWAT M.: Accelerating spatial data visualization dashboards via a materialized sampling approach. In *Proceedings of the International Conference on Data Engineering, ICDE* (2020). 2

[lea] Leaflet. https://leafletjs.com/. 1

[LH14] LIU Z., HEER J.: The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph. 20*, 12 (2014), 2122–2131. 1

[LKS13] LINS L. D., KLOSOWSKI J. T., SCHEIDEGGER C. E.: Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph. 19*, 12 (2013), 2456–2465. 2, 3

[LWSY20] LIU C., WU C., SHAO H., YUAN X.: Smartcube: An adaptive data management architecture for the real-time visualization of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph. 26*, 1 (2020), 790–799. 2

[map] Mapbox. https://www.mapbox.com/. 1

[MCW*20] MEI H., CHEN W., WEI Y., HU Y., ZHOU S., LIN B., ZHAO Y., XIA J.: Rsatree: Distribution-aware data representation of large-scale tabular datasets for flexible visual query. *IEEE Trans. Vis. Comput. Graph. 26*, 1 (2020), 1161–1171. 2

[MFDW17] MORITZ D., FISHER D., DING B., WANG C.: Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017.* (2017), pp. 2904–2915. 2

[PCM16] PARK Y., CAFARELLA M. J., MOZAFARI B.: Visualization-aware sampling for very large databases. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016* (2016), pp. 755–766. 2

[pix] Pixel bitmap library. https://github.com/ISG-ICS/rainbow/blob/main/server/public/javascripts/lib/webgl_point_layer.js. 1

[PSC00] PAPPAS T. N., SAFRANEK R. J., CHEN J.: Perceptual criteria for image quality evaluation. *Handbook of image and video processing* (2000), 669–684. 2, 4

[PSF12] PLANTHABER G., STONEBRAKER M., FREW J.: Earthdb: scalable analysis of MODIS data using scidb. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2012, Redondo Beach, CA, USA, November 6, 2012* (2012), Chandola V., Vatsavai R. R., Gupta C., (Eds.), ACM, pp. 11–19. 2

[Sch14] SCHEIBLAUER C.: *Interactions with gigantic point clouds*. PhD thesis, 2014. 2

[Sch21] SCHÜTZ M.: *Interactive exploration of point clouds*. PhD thesis, Wien, 2021. 2

[SOW20] SCHÜTZ M., OHRHALLINGER S., WIMMER M.: Fast out-of-core octree generation for massive point clouds. *Comput. Graph. Forum 39*, 7 (2020), 155–167. 2

[TdLPRC18] TOSS J., DE LARA PAHINS C. A., RAFFIN B., COMBA J. L. D.: Packed-memory quadtree: A cache-oblivious data structure for visual exploration of streaming spatiotemporal big data. *Comput. Graph. 76* (2018), 117–128. 2

[TLD*19] TAO W., LIU X., DEMIRALP Ç., CHANG R., STONEBRAKER M.: Kyrix: Interactive visual data exploration at scale. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings* (2019). 2

[WBS*04] WANG Z., BOVIK A. C., SHEIKH H. R., SIMONCELLI E. P., ET AL.: Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing 13*, 4 (2004), 600–612. 2

[WCLY15] WANG L., CHRISTENSEN R., LI F., YI K.: Spatial online sampling and aggregation. *PVLDB 9*, 3 (2015), 84–95. 2

[WLX19] WANG B., LI M., XIONG L.: Fastgeo: Efficient geometric range queries on encrypted spatial data. *IEEE Trans. Dependable Secur. Comput. 16*, 2 (2019), 245–258. 2

[YMS17] YU J., MORAFFAH R., SARWAT M.: Hippo in action: Scalable indexing of a billion new york city taxi trips and beyond. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017* (2017), IEEE Computer Society, pp. 1413–1414. 2

[YZS18] YU J., ZHANG Z., SARWAT M.: Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem. In *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018* (2018), pp. 15:1–15:12. 2

© 2022 The Author(s)
Eurographics Proceedings © 2022 The Eurographics Association.