Dynamic Reduction of Query Result Sets for Interactive Visualizaton

Leilani Battle[*] Massachusetts Institute of Technology
Remco Chang[†] Tufts University
Michael Stonebraker[‡] Massachusetts Institute of Technology

Abstract

Modern database management systems (DBMS) have been designed to efficiently store, manage and perform computations on massive amounts of data. In contrast, many existing visualization systems do not scale seamlessly from small data sets to enormous ones. We have designed a three-tiered visualization system called ScalaR to deal with this issue. ScalaR dynamically performs resolution reduction when the expected result of a DBMS query is too large to be effectively rendered on existing screen real estate. Instead of running the original query, ScalaR inserts aggregation, sampling or filtering operations to reduce the size of the result. This paper presents the design and implementation of ScalaR, and shows results for an example application, displaying satellite imagery data stored in SciDB as the back-end DBMS.

Index Terms: H.2.4 [Database Management]: Systems—Query Processing; H.2.8 [Database Management]: Database applications—Data mining

1 Introduction

Since their introduction by Ahlberg, Williamson and Shneiderman in 1992 [4], dynamic queries have made their way into many visual data exploration systems. Projects such as Polaris[14]/Tableau[2], Spotfire[3], VisDB[9] and DEVise[10], focus on visual exploration of databases. Similarly, projects like Xmdv Tool[15], VQE[7], Visage[12], and Improvise[16] implement dynamic queries over data loaded from files. However, as analytical and scientific data sets continue to increase in magnitude, visual analytics systems will need to incorporate scalable data management solutions to remain interactive. Otherwise, interactive visual analytics of large-scale data will remain difficult if not impossible.

Unfortunately, many information visualization systems do not scale seamlessly from small data sets to massive ones. In these systems, a common bottleneck is the amount of data that needs to be transferred from the database management system (DBMS) to the front-end visualization. In addition, as information visualization systems draw more and more query results on the screen, the result is excessively dense graphs that are difficult to interpret. In order to provide useful visualizations of large-scale data sets efficiently, existing systems need to summarize large data sets on the fly. Without automatic summaries, the user is forced to write queries by hand, or manually filter through the system's dynamic querying interface to reduce the data set down to the desired size. To address these issues, several systems like Table Lens[11] and the Attribute Explorer[13] automatically provide summaries through their visualizations. Alternatively, sampleAction[8] explores progressive querying of DBMS's, visualizing incremental query results with error bounds so the user can stop execution when they've reached their desired accuracy level.

[*]e-mail: leibatt@mit.edu
[†]e-mail: remco@cs.tufts.edu
[‡]e-mail: stonebraker@csail.mit.edu

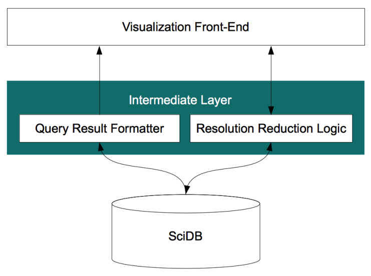


Figure 1: ScalaR system architecture.

In our paper, we address this problem for large-scale data by performing resolution reduction on query results. Our approach dynamically determines if the result of a DBMS query is too large to be effectively rendered on existing screen real estate. Based on this information, we use the DBMS to reduce the visualization space by inserting aggregation, sampling and/or filtering operations into the query. To demonstrate our approach, we developed a three-tiered prototype visualization system called ScalaR, and provide results from an example application of ScalaR, displaying satellite imagery data stored in SciDB[6] as the back-end DBMS.

2 Architecture

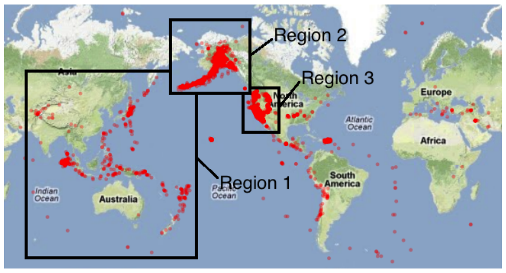
The design principle of ScalaR's architecture is to allow for "thin" visualizations that are oblivious to the size of the data. Therefore ScalaR can visualize large amounts of data on commodity hardware. ScalaR has 3 major components: a web front-end responsible for taking user input and drawing the visualizations with the resulting data; an intermediate layer on the server that takes input from the front-end and translates it into queries for the DBMS to execute, incorporating resolution reduction as necessary; and a DBMS that executes the queries dispatched from the intermediate layer. Figure 1 shows the layout of ScalaR architecture. The three major architecture components are described in more detail below.

2.1 Web Front-End

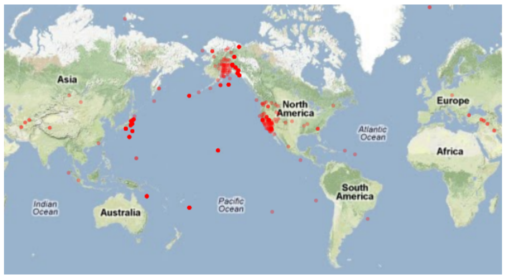
We implemented a web-based front end, using the D3.js [5] Javascript library to draw the visualizations. The front end supports the following visualizations: scatterplots, line charts, histograms, mosaic plots, heat maps, and tree maps. The user inputs a query into a text box on the screen and selects a visualization type through a drop-down menu. After choosing the visualization, the user is given a second set of optional menus to specify characteristics of the visualization. For example, what attributes in the query results correspond to the x and y axes. ScalaR's architecture supports pan and zoom functionality, both of which trigger new dynamic queries over the DBMS to retrieve missing data as the user explores the data set.

2.2 Intermediate Layer

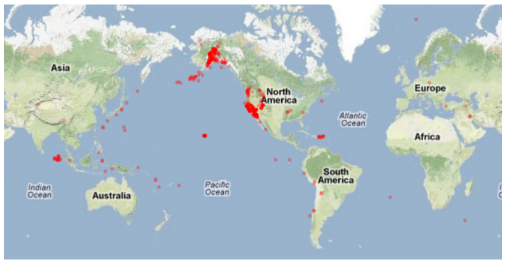
The intermediate layer consists of server code that takes user queries from the front-end, dispatches queries to the DBMS, and



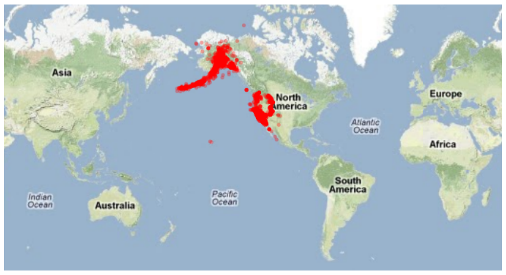
(a) Original query, with marked regions of interest



(b) Aggregation



(c) Sampling



(d) Filtering

Figure 2: Map plots for a query manipulated using several resolution reduction techniques.

prepares the resulting data for consumption by the front-end. Before dispatching user-defined queries to the DBMS, the intermediate layer retrieves the proposed query plan from the DBMS and uses this information to compute the expected size of the result. The intermediate layer uses this calculation to decide whether to insert a resolution reduction operation into the original query. See Section 3 for a list of resolution reduction operations.

2.3 DBMS

SciDB[6] is the DBMS used in ScalaR. SciDB is geared towards managing large-scale array-based data, particularly for data sets that require complicated computations. SciDB has several native functions that make the query rewriting process in the intermediate layer simpler. For example, the `bernoulli` function performs uniform sampling over the data. The `regrid` function divides the data into a grid of subarrays and returns summaries of the subarrays, such as averages or counts.

SciDB stores data as multi-dimensional matrices. Users specify the dimensions of the matrix, and the attributes of each element in the matrix. SciDB supports two languages for querying the data: Array Functional Language (AFL), or the SQL-like language Array Query Language (AQL). When writing queries, the attributes and dimensions can be thought of as columns and elements in the matrix as tuples. The combined dimensions act as keys for elements in the matrix (see Section 3.2 for examples of SciDB queries).

3 Resolution Reduction

3.1 Problem Definition and Approach

There are two issues many existing visualization systems face when drawing very large data sets. First, if the system is only designed to handle data sets up to a certain size, the system will spend all of its time trying to manage the visualization. The system may even "freeze" for an extended period of time or crash while trying to draw the visualization. The second issue is that even if the visualization system can draw arbitrarily large results, there may be so many objects to draw on the screen that the resulting visualization is too dense to be useful to the user.

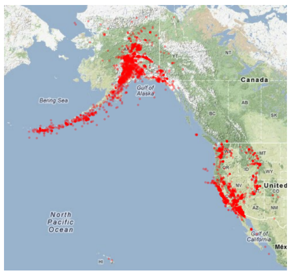


Figure 3: Zoom on regions 2 and 3 over filtered query results.

We have implemented several Resolution Reduction techniques to resolve these issues automatically. They can be applied to the data directly, as well as objects to be drawn in the visualization. The techniques are as follows:

Aggregation: Given a summary operation, the data is grouped into submatrices, and summaries of the submatrices are returned as the result. Summary operations include: sum, average, max and min.

Sampling: Given a probability value *p*, return roughly that fraction of data as the result. Most databases already support this operation.

Filtering: Given a set of desired characteristics of the data, return the elements that meet these characteristics. These characteristics are translated into WHERE clause predicates.

The final query results are sent to the front-end as a list of objects, where each object in the list stores the attribute and dimension values of one cell from the resulting array. The resolution reduction techniques can also be applied to the formatted list. For example, the intermediate layer may return a uniform sampling of the objects, or a grouping of the objects based on a given attribute/dimension of the data. Performing resolution reduction on the formatted list provides finer granularity for resolution reduction that is independent of the storage model of the DBMS.

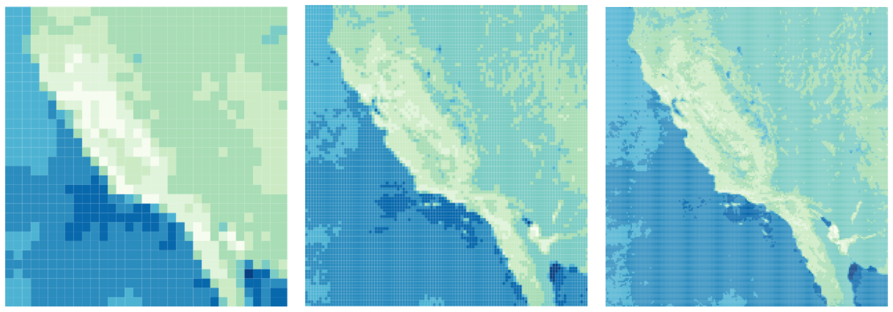

(a) 1000 points resolution (b) 10000 points resolution (c) 40000 points resolution

Figure 5: Zoom on the California region of the `ndvi_points` array at 1000, 10000, and 40000 points resolution

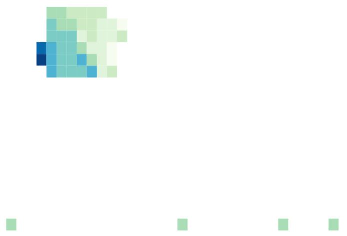


Figure 4: Overview visualization of the `ndvi_points` array

3.2 Example

Suppose a user of the ScalaR system wants to plot earthquake data to see the distribution of earthquakes around the world. She inputs the following query, and requests a map plot of the results: `select latitude, longitude from quake`. Figure 2a shows the resulting plot. ScalaR uses the Google Maps API[1] to create map plots. The distribution of earthquakes appears to be evenly divided between 3 regions, specified in Figure 2a. However, two important details are hidden from the user due to over-plotting: the largest region actually contains only a small percentage of earthquakes, and there are considerably more earthquakes around California than Alaska. The rest of this section explains in detail how the plot in Figure 2a is misleading, and provides alternative plots for the data using the techniques in the previous section.

The user has a 6381 by 6543 sparse array containing records for 7576 earthquakes. The data is stored in SciDB with the following schema: `quake(datetime, magnitude, depth, latitude, longitude, region)[lat, lon]`. Array attributes are listed in the parentheses, followed by dimensions in brackets. Note that longitude and latitude coordinates are represented twice in the schema. SciDB provides operations for transforming attributes into new dimensions, such as the latitude and longitude attributes in `quake`. However, SciDB replaces the original values of non-integer attributes with a zero-based integer range. To maintain access to the original values, the latitude and longitude coordinates were also stored as attributes in the array.

The user picks 3 major regions of interest in this plot, identified by the three boxes drawn in Figure 2a. Region 1covers Asia and Australia. Region 2 is the Alaska area, and region 3 is the west coast of the US excluding Alaska. According to Figure 2a, region 1 appears to contain at least 25% of the plotted earthquakes. In addition, the points in region 2 cover a larger area of the plot, so region 3 seems to have less seismic activity than region 2. However, this plot is misleading. All 7576 earthquakes are plotted, but over-plotting obscures the true densities of these three regions. Ignoring overlap with region 2, region 1 actually contains only 548 points, or less than 8% of all plotted earthquakes. Regions 2 and 3 have 2423 points (over 30%) and 4081 points (over 50%), respectively. Thus region 3 actually contains over 50% more points than region 2.

Now suppose we reduce the resolution to roughly 500 points to provide the user with a clearer view of the distribution. Aggregation over the array produces the following query: `select avg(latitude), avg(longitude) from (select latitude, longitude from quake) regrid 289, 289`, where the original query is wrapped in a SciDB `regrid` statement. This query tells SciDB to divide quake into 289 by 289 subarrays along the lat and lon dimensions, and summarize the subarrays by taking the average of the latitude and longitude coordinates within in each subarray. ScalaR makes all dimensions of the subarrays the same width, and computes this value using the following function: $\lceil \sqrt[d]{\frac{|\text{quake}|}{t}} \rceil$, where d is the number of dimensions, and t is a computed threshold value. In this example, $t = 500$. The resulting array has 529 cells, and Figure 2b shows the resulting plot. Region 1 now contains much fewer points, and the majority of points are clearly located in regions 2 and 3. In addition, the plotted points in region 2 are spread out and visibly less dense than the points plotted in region 3. Despite some loss of detail due to the reduction in points, but the distribution of earthquakes is accurately characterized by the aggregation plot.

Sampling to reduce the resolution produces the following query: `select latitude, longitude from bernoulli( (select latitude, longitude from quake), 0.0659978881)`, where the original query is wrapped in a SciDB `bernoulli` statement. This query tells SciDB to randomly choose points from quake, where each point is chosen with probability 0.0659978881. ScalaR computes this probability by dividing t by the total number of non-empty cells in `quake`. Figure 2c shows a plot of the result, which returns 526 points. Like aggregation, sampling results in few points drawn in region 1, and the majority of points drawn in regions 2 and 3. More points are drawn in the areas that were densest in the original plot, providing a concise summary of the distribution of earthquakes.

Now that the user has identified the regions with the most earthquakes, she can use filtering to reduce the resolution of the data in favor of these regions. This results in the following query to retrieve points in regions 2 and 3: `select latitude, longitude from quake where lat > 20 and (lon < -100 or lon > 170)`. Figure 2d shows the resulting plot. She can then zoom into regions 2 and 3 to see the distribution of earthquakes in more detail, as shown in Figure 3.

4 Visualizing Satellite Image Data

We implemented an example application that visualizes query results for normalized difference vegetation index (NDVI) calculations over a subset of NASA satellite imagery data. The data set was roughly 27GB in size, covered the state of Cali-

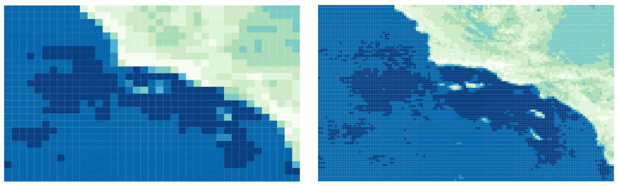

(a) 1000 points resolution (b) 10000 points resolution

Figure 6: Zoom on LA area at 1000 and 10000 points resolution

fornia, and was stored in a single, two-dimensional sparse matrix called `ndvi_points` in SciDB. The schema was as follows: `ndvi_points(ndvi)[longitude,latitude]`. The latitude and longitude coordinates were used to dimension the array, and the ndvi calculations were stored as an attribute of the array. The NDVI calculations were visualized as heatmaps, and aggregation was used to reduce the resolution of the data.

Consider the scenario where the user wants an overview of the of the NDVI data over the southern California coast, primarily from Santa Barbara to San Diego. The user starts by inputting a query to retrieve all the NDVI calculations from the array: `select ndvi from ndvi_points`. Without resolution reduction, this query returns over one billion points. In addition, the actual dimension ranges of the array are on the order of millions, which would result in a sparse heatmap with over one trillion cells. This is clearly too large of an image to draw on the screen, so ScalaR prompts the user to reduce the resolution. Using aggregation, ScalaR produces an initial visualization at a resolution of about 1000 points, shown in Figure 4. Resolution refers to the size of the query results being drawn, so Figure 4 shows the result of reducing the data down to a 33 by 33 matrix. This visualization clearly shows the sparseness of `ndvi_points`, and reveals a dense area of data in the array.

Now the user tells the application to zoom in on the dense portion of the array using a selection box. The resulting visualization at a resolution of 1000 points is shown in Figure 5a. The general shape of the western coast of California/Northern Mexico is apparent, but the user may want the image to be clearer. Figures 5b and 5c show the results of increasing the resolution to 10000 and 40000 points respectively, where the identity of the region is very clear in both images. The user can now clearly identify the desired southern California region, and zooms in to the Los Angeles, Santa Barbara area as shown in Figure 6.

To perform the same tasks without ScalaR, the user would have to write aggregation queries manually over the data set. She has to manually identify the desired region of the array to visualize, and perform her own calculations to determine a reasonable resolution for the results. She may also need to store the query results in a separate file to load into her desired visualization system. The user also resorts to trial and error, potentially repeating the above steps many times before finding her desired region and resolution for the image. ScalaR eliminates the need to manually write queries to reduce the resolution of the data, providing the user with more information quickly and easily.

5 Conclusions and Future Work

We presented the design and implementation of ScalaR, an information visualization system that dynamically performs resolution reduction when expected query results from the back-end DBMS are too large to be effectively drawn on the screen. ScalaR uses aggregation, filtering and/or sampling queries to downsize query results, providing the user with more or less detail as necessary to reduce completion time and maximize usage of existing screen real estate. We presented results for an example application of ScalaR, visualizing satellite imagery data stored in SciDB.

We plan to make several optimizations in ScalaR's design, starting with the 2 following approaches. The first is to push optimizations into the database by improving query optimization, and by writing queries to utilize information from the front-end about what is being drawn. Second, we plan to incorporate prefetching in the middle layer of our architecture, using feedback from the front-end about user interactions; for example, whether the user just zoomed in, or the direction the user is panning through the visualization.

References

[1] Google maps api. `https://developers.google.com/maps/`, May 2012.

[2] Tableau software. `http://www.tableausoftware.com/`, May 2012.

[3] Tibco spotfire. `http://spotfire.tibco.com/`, May 2012.

[4] C. Ahlberg, C. Williamson, and B. Shneiderman. Dynamic queries for information exploration: an implementation and evaluation. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '92, pages 619–626, New York, NY, USA, 1992. ACM.

[5] M. Bostock, V. Ogievetsky, and J. Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[6] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A demonstration of scidb: a science-oriented dbms. *Proc. VLDB Endow.*, 2(2):1534–1537, Aug. 2009.

[7] M. Derthick, J. Kolojejchick, and S. F. Roth. An interactive visualization environment for data exploration. In *IN PROC. OF KNOWLEDGE DISCOVERY IN DATABASES*, pages 2–9. Press, 1997.

[8] D. Fisher, I. Popov, S. Drucker, and m. schraefel. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, CHI '12, pages 1673–1682, New York, NY, USA, 2012. ACM.

[9] D. Keim and H.-P. Kriegel. Visdb: database exploration using multidimensional visualization. *Computer Graphics and Applications, IEEE*, 14(5):40 –49, sept. 1994.

[10] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: integrated querying and visual exploration of large datasets. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 301–312, New York, NY, USA, 1997. ACM.

[11] R. Rao and S. K. Card. The table lens: merging graphical and symbolic representations in an interactive focus + context visualization for tabular information. In *Proceedings of the SIGCHI conference on Human factors in computing systems: celebrating interdependence*, CHI '94, pages 318–322, New York, NY, USA, 1994. ACM.

[12] S. F. Roth, P. Lucas, J. A. Senn, C. C. Gomberg, M. B. Burks, P. J. Stroffolino, A. J. Kolojechick, and C. Dunmire. Visage: a user interface environment for exploring information. In *Proceedings of the 1996 IEEE Symposium on Information Visualization (INFOVIS '96)*, INFOVIS '96, pages 3–, Washington, DC, USA, 1996. IEEE Computer Society.

[13] R. Spence and L. Tweedie. The attribute explorer: information synthesis via exploration. *Interacting with Computers*, 11(2):137 – 146, 1998.

[14] C. Stolte and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In *Proceedings of the IEEE Symposium on Information Vizualization 2000*, INFOVIS '00, pages 5–, Washington, DC, USA, 2000. IEEE Computer Society.

[15] M. O. Ward. Xmdvtool: integrating multiple methods for visualizing multivariate data. In *Proceedings of the conference on Visualization '94*, VIS '94, pages 326–333, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[16] C. Weaver. Building highly-coordinated visualizations in improvise. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 159–166, Washington, DC, USA, 2004. IEEE Computer Society.