Dynamic Reduction of Query Result Sets for Interactive Visualization

Leilani Battle, Michael Stonebraker
Electrical Engineering and Computer Science Department
MIT

Remco Chang
Department of Computer Science
Tufts University

***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.**

***Keywords*-data analysis; scientific computing; interactive visualization**

I. Introduction

Modern database management systems (DBMS) are designed to efficiently store, manage and perform computations on massive amounts of data. In addition, scientific data sets are growing rapidly to the point that they do not fit in memory. As a result, more analytics systems are relying on databases for the management of big data. For example, many popular data analysis systems, such as Tableau [1], Spotfire [2], R and Matlab, are actively used in conjunction with database management systems. Furthermore, Bronson *et. al.* [3], show that distributed data management and analysis systems like Hadoop [4] have the potential to power scalable data visualization systems.

Unfortunately, many information visualization systems do not scale seamlessly from small data sets to massive ones by taking advantage of these data management tools. Current workflows for visualizing data often involve transferring the data from the back-end database management system(DBMS) to the front-end visualization system, placing the burden of efficiently managing the query results on the visualizer.

To avoid this, many large-scale visualization systems rely on fitting the entire data set within memory, tying data analytics and visualization directly to the management of big data. However, this limits adoptability of these systems in the real world, and draws the focus of these systems away from producing efficient and innovative visualizations for scientific data and towards general storage and manipulation of massive query results.

To address these issues, we developed a flexible, three-tiered scalable interactive visualization system for big data named ScalaR, which leverages the computational power of modern database management systems to power the back-end analytics and execution. ScalaR relies on query plan estimates computed by the DBMS to perform resolution reduction, or how to dynamically determine how to summarize massive query result sets on the fly. We provide more details on resolution reduction below

ScalaR places a limit in advance on the amount of data the underlying database can return. This data limit can be driven by various performance factors, such as the resource limitations of the front-end visualization system.We insert a middle layer of software between the front-end visualization system and underlying database management system (DBMS) that dynamically determines when a query will violate the imposed data limit and delegates to the DBMS how to reduce the result accordingly.

We rely on statistics computed by the DBMS to quickly compute necessary reduction estimates by analyzing query plans (see Section III-A for more information about query plans). This decouples the task of visualizing the data from the management and analysis of the data. This also makes our design back-end agnostic, as the only requirements are that the back-end must support a query API and provide access to metadata in the form of query plans. To demonstrate our approach, we provide use-cases visualizing earthquake records and NASA satellite imagery data in ScalaR using SciDB as the back-end DBMS.

In this paper, we make the following contributions:

- We present a modularized approach to query result reduction using query plans for limit estimation and leveraging native database operations to reduce query results directly in the database.
- We present the architechture for a scalable information visualization system that is completely agnostic to the underlying data management back-end.
- We present motivating examples for ScalaR using earthquake data and NASA MODIS satellite imagery data.
- We present initial performance results for using ScalaR to visualize NASA MODIS satellite imagery data.

A. Related Work

In an effort to push data management outside of the visualization system and into the back-end DBMS, several existing techniques and systems provide functionality for reducing the amount data visualized. For example, Jerding *et. al.* [5] compress entire information spaces into a given pixel range using pixel binning, and color cues to denote pixel overlap in the reduced visualization. Elmqvist *et. al.* [6] use hierarchical aggregation to reduce the underlying data and reduce the number of elements drawn in the visualization. Hierarchical aggregation transforms visualizations into scalable, multi-level structures that are able to support multiple resolutions over the data.

Another prevalent technique for data reduction in data analytics and visual analysis is building OLAP cubes to summarize data [7]. To produce OLAP cubes, the underlying data is binned and simple statistical calculations, such as count and average, are computed over the bins. Liu *et. al.* use this technique in the imMens system to reduce data in the back-end DBMS, which combined with front-end WebGL optimizations allows imMens to draw billions of data points in the web browser.

Hellerstein *et. al.* [8], [9], [10] present an alternative approach to data reduction through incremental, progressive querying of databases. Progressive querying initially samples a small set of the data to quickly produce a low-accuracy result. Over time, the database samples more data to improve the accuracy of the result. Users can wait for the query to finish for complete results, or stop execution early when the result has reached their desired error bounds. Fisher *et. al.* [11] revisit this approach in greater detail with simpleAction, focusing on applying iterative query execution to improve interactivity of database visualizers. simpleAction visualizes incremental query results with error bounds so the user can stop execution when they've reached their desired accuracy level. Instead of waiting for the user to specify when to stop execution, Agarwal *et. al.* present a different approach to fast approximate query execution in their BlinkDB [12] system. BlinkDB executes queries over stratified samples of the data set built at load time, and also provides error bounds for query results.

ScalaR provides data-reduction functionality that is similar to the systems described above. However, ScalaR also provides functionality to specify a limit on the amount of data the DBMS can return in advance, and dynamically modifies the reductions accordingly. This allows the front-end to specify more or less data on-the-fly, with minimal knowledge of the back-end DBMS. As a result, ScalaR provides more flexibility in choosing techniques for data reduction, as reduction techniques can be added to the back-end without modifying the front-end. In addition, the user is not responsible for building summaries or samples of the data in advance, thus reducing the level of expertise required to manage the back-end DBMS for visualization with ScalaR.

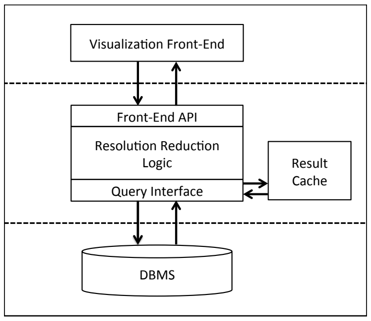


Figure 1. ScalaR system architecture.

II. Architecture

ScalaR has 3 major components: a web-based front-end, a middle layer between the front-end and DBMS, and SciDB as the back-end DBMS (see Figure 1). They are described in detail below.

A. Web Front-End

We implemented a web-based front end, using the D3.js [13] Javascript library to draw the visualizations. ScalaR supports scatterplots, line charts, histograms, map plots and heat maps. The Google Maps API [14] is used to draw map plots. The user inputs a query into a text box on the screen and selects a visualization type through a drop-down menu. The user can also specify a *resolution* for the result, *i.e.* how many total data points they want the query result to return, via 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.

B. Intermediate Layer

The intermediate layer consists of server code that takes user queries from the front-end, dispatches queries to the DBMS, and 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 III-B for a list of resolution reduction operations. Query results are stored in a result cache for future use. A straight-forward eviction policy, such as LRU, is used to remove old data from the cache.

C. DBMS

Due to it's ease of use with scientific data sets, SciDB [15] is the primary back-end DBMS used in ScalaR. SciDB is geared towards managing large-scale array-based data. Users specify the dimensions of the matrix, and the attributes of each element in the matrix (see Section IV-A for examples of SciDB queries). However, ScalaR is database agnostic in design, and can be extended to support a variety of data management systems.

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 IV-A for examples of SciDB queries).

SciDB is selected as the primary DBMS for ScalaR because the array-based structure affords some native operations that are fast and efficient for typical visualization tasks with large scientific data set. Basic resolution reduction techniques, such as filtering and sampling, can be implemented directly for most database management systems with native operations.

III. Resolution Reduction

In this section, we describe the general resolution reduction techniques used to develop ScalaR, and how they are implemented using SciDB as the back-end DBMS.

A. Retrieving Metadata From the Query Optimizer

Almost all DBMS have a query compiler, which is usually responsible for parsing, interpreting and generating an efficient execution plan for queries. The query compiler usually includes a component called the query optimizer, which is the principal unit for improving query performance. Metadata must be calculated in order for the query optimizer to produce accurate estimates for query plans over the given data set. This metadata includes statistics about the data set and other useful information, such as the estimated number of cells or tuples to be produced by the query.

Modern DBMS are designed to produce query plans very cheaply in terms of time and resources. Statistics and other various properties of the data are calculated and stored when the data is loaded and during the execution of queries. Query optimization in itself is a very important research problem within the database community, and is thus outside the scope of this paper. However, it is hopefully clear that executing commands to retrieve query plans is significantly cheaper than executing expensive queries over gigabytes of data or more.

Most DBMS expose some amount of metadata from the query optimizer to users in the form of special commands. For example, PostgreSQL provides this functionality via the `EXPLAIN` command, which provides the user with query plan information. SciDB exposes query plan information through the `explain_logical` and `explain_physical` commands.

Suppose we have an earthquake data set stored in SciDB in an array named `earthquake`, and you want to see how SciDB will execute the same query:

```
scan(earthquake);
```

The scan operation in SciDB is the same as "`SELECT *`" syntax in relational databases. To generate a query plan, we execute the following in SciDB:

```
explain_physical('scan(earthquake)','afl');
```

This produces the following query plan:

```
[("[pPlan]:
  schema earthquake
    <datetime:datetime NULL DEFAULT null,
    magnitude:double NULL DEFAULT null,
    latitude:double NULL DEFAULT null,
    longitude:double NULL DEFAULT null>
    [x=1:6381,6381,0,y=1:6543,6543,0]
  bound start {1, 1} end {6381, 6543}
  density 1 cells 41750883 chunks 1
  est_bytes 7.97442e+09
")]
```

The whole schema of the resulting array is provided beginning on line two of the query plan. The dimensions of the array are `x`, and `y`, given by "`x=1:6381`" and "`y=1:6543` in the dimension description. We also see from this dimension description that the resulting array will be 6381 by 6543 in dimension. SciDB also provides the bounds of the array explicitly in "`bound start 1, 1 end 6381, 6543`". The attributes of the array are provided on the lines just before the dimension description: `datetime`, `magnitude`, `latitude`, and `longitude`. SciDB attributes are similar to columns in relational databases. The number of cells in the array is 41750883, given by "`cells 41750883`" on the last line of the query plan. The number of SciDB chunks used by the array (SciDB's unit of storage on disk) is 1, given by "`chunks 1`" on the same line. The estimated number of bytes to store the result is given by "`est_bytes 7.97442e+09`".

Query plans are essential to databases because they provide valuable information about how the query will be executed, and help the database reason about the relative cost of various query operations. For the user, query plans provide insight and additional information about the query that is very difficult for humans to reason about without any prior experience with the data set or previous query results to reference. Lastly, this information costs very little to retrieve from a modern DBMS compared to executing the query directly, especially when working with massive data sets.

B. General Resolution Reduction Techniques

There are two issues many existing visualization systems face when drawing very large data sets. Current systems have to spend a considerable amount of time managing data, which becomes increasingly problematic with more and more data. Also, these systems lack effective tools for automatically aggregating results. Therefore, there may be so many objects to draw on the screen that the resulting visualization is too dense to be useful to the user.

There are two commonly-used approaches to handling large amounts of data stored in a DBMS that we have automated, sampling a subset of the data or aggregating the data (*i.e.* `GROUP BY` queries). When the data set is dense, aggregation significantly reduces the resulting number of points by grouping points by proximity. When the data is sparse, it is difficult to gauge the distribution of data across the array. Sampling results in a subset of data of predictable size that is independent of the distribution of data in the underlying array. When criteria is known for identifying non-relevant data, filtering the data directly is also an option.

Each reduction technique takes an implicit parameter n specified by the Intermediate Layer that is adjusted based on the desired result set size. The techniques are as follows:

- Aggregation: Group the data into n sub-matrices, and return summaries over the sub-matrices. Summary operations include: sum, average, and max/min.
- Sampling: Given a probability value p, return roughly that fraction of data as the result, where $p * |\text{data}| = n$. Most databases already support this operation.
- Filtering: Given a set filters over the data, return the elements that pass these filters. These filters are translated into WHERE clause predicates.

The rest of this section describes in detail how ScalaR's intermediate layer retrieves and analyzes query metadata from the DBMS and manages resolution reduction.

C. Analyzing SciDB Query Plans for Resolution Reduction

We describe in this section how the intermediate layer estimates resolution reduction calculations with SciDB as the back-end DBMS.

When ScalaR's front-end receives a query and desired resolution from the user, this information is first passed to the intermediate layer. ScalaR's intermediate layer then requests query plan information for the user's query from the DBMS using the commands described in Section III-A. ScalaR extracts the estimated size of the query result from the resulting query plan information, and compares this value to the user's desired resolution. If the estimated size is larger than the resolution value, the intermediate layer sends a response to the front end indicating that the estimated size of the result is larger than the user's desired resolution.

The front-end then notifies the user that the result will be "too big", and gives the user the option of choosing a resolution reduction approach to produce a smaller result, or to return the full result anyway without any form of resolution reduction. See Section III-B for more information on resolution reduction techniques. Note that ScalaR is estimating using only query plan information at this point, and no queries have been performed on the actual data set.

If the user decides not to reduce the result, the intermediate layer dispatches the user's original query for execution on the database, formats the results, and returns the formatted results to the front-end for visualization.

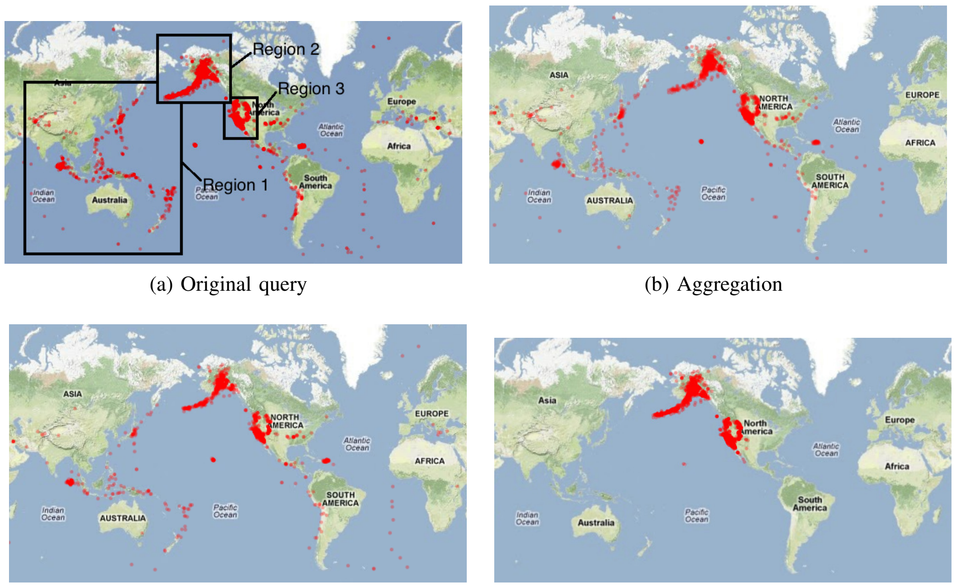
If the user chooses a resolution reduction technique, ScalaR performs estimation calculations *before* sending any queries to the DBMS, and thus no costly operations need to be performed on the original data set while the intermediate layer is constructing the final query incorporating resolution reduction.

1) Aggregation: Given a d-dimensional SciDB array A and desired resolution n, ScalaR aggregates over A by dividing A into at most n d-dimensional sub-arrays, performing a summary operation over all sub-arrays, and returning the summary results. Examples of summary operations over the sub-arrays include taking the sum, average or standard deviation across all elements in the sub-array.

As described in Section II-C, SciDB already has a native operation called `regrid` that will perform aggregation automatically. However, SciDB does not take the number of desired sub-arrays n as input, and instead requires the desired dimensions of the sub-arrays. For example, to divide a 2-dimensional 16 by 16 array into 16 sub-arrays using `regrid`, ScalaR needs to specify sub-array dimensions such that each sub-array contains 16 elements each. This can be achieved by setting the sub-array dimensions to be 4 by 4. Sub-array dimensions of 2 by 8 or 1 by 16 will also result in 16 sub-arrays total. Note that dimensions in SciDB have a specific order, so the ordering of the sub-array widths matters. For example, using 2 by 8 sub-arrays will not produce the same result as using 8 by 2 sub-arrays.

To reduce A to the desired user resolution n, ScalaR needs to aggregate over A to create A' such that $|A'| \leq n$. The simplest approach is to assume that the same number of sub-arrays should be generated along every dimension. To do this, ScalaR first computes the dth root of n, which we refer to as n_d. ScalaR then computes s_i, or the sub-array width along dimension i, for all dimensions i by dividing the width of A along dimension i by n_d.

2) Sampling: When a user requests that data be sampled to reduce the resolution, ScalaR returns a uniform sample of the result. Most DBMS already provide their own uniform sampling operations. SciDB's `bernoulli` function performs uniform sampling over a given array A with sampling rate p and seed, where $0 \leq p \leq 1$. The seed used is a default global variable chosen by us.



(a) Original query (b) Aggregation

(c) Sampling (d) Filtering

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

To reduce A to the desired user resolution n, ScalaR needs to sample over A to create A' such that $|A'| \leq n$. ScalaR computes the sampling rate p as the ratio of resolution n to total array elements $|A|$, or $p = \frac{n}{|A|}$. If the resulting number of points in the sampled A' is greater than n, ScalaR randomly removes $|A'| - n$ points from A'.

3) Filtering: Currently, the user specifies explicitly via text what filters to add to the query. These filters are translated into SciDB `filter` operations. Note that extensive work has already been done in creating dynamic querying interfaces, where users can specify filters without writing their own SQL queries. Thus it is straightforward to extend ScalaR's front-end to incorporate a dynamic querying interface for specifying filters in a more intuitive way.

IV. Motivating Examples

We now present two use cases that demonstrate how ScalaR addresses the issues presented in Section I.

A. Earthquake Data

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.
```

The user has stored in SciDB a 6381 by 6543 sparse array containing records for 7576 earthquakes. The schema is as follows:

```
quake(datetime, magnitude, depth, latitude,
  longitude, region)[x,y]
```

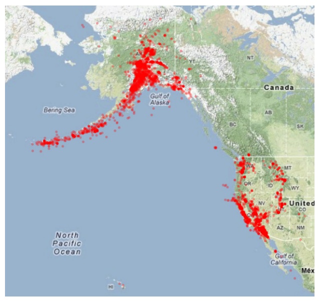


Figure 3. Zoom on regions 2 and 3 over filtered query results.

Array attributes are listed in the parentheses, followed by dimensions in brackets. The dimensions x and y represent a 2-dimensional mesh of the latitude and longitude coordinates to take advantage of spatial locality when storing the data points as a SciDB array. Note also that every record in this example data set has a unique pair of latitude and longitude points for convenience.

The user picks 3 major regions of interest in this plot, identified by the three boxes drawn in Figure 2a. Region 1 covers Asia and Australia. Region 2 is the Alaska area, and region 3 is the west coast of the US excluding Alaska. Figure 2a shows a classic over-plotting problem with map visualizations, where each recorded earthquake is represented as a red dot. Region 1 in Figure 2a 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. Region 2 has 2423 points (over 30%), and region 3 4081 points (over 50%). Thus region 3 actually contains over 50% more points than region 2.

This scenario lends itself to two separate goals for resolution reduction. If the physical over-plotting of points on the map is the motivating factor, the reduction can be driven by the width and height of the visualization canvas. As in the case of this example, the volume of data being returned by the back-end DBMS can also be motivation for resolution reduction, which affects many performance-related factors, such as limited bandwidth, latency, and rendering speed.

Now suppose we ask ScalaR to reduce the matrix size of `quake` from 6381 by 6543 to 40000 maximum using aggregation. ScalaR first takes the dth root of n to compute the number of subarrays along every dimension n_d (see Section III-C for more information):

$$n_d = \lfloor \sqrt{40000} \rfloor$$

where d is the number of dimensions (2) and n is our desired resolution (40000). n_d is 200in this example. ScalaR then computes the width of each dimension i of the sub-arrays by dividing the original width of i by n_d:

$$s_1 = \lceil 6381/200 \rceil, s_2 = \lceil 6543/200 \rceil$$

In this example, $s_1 = 32$ and $s_2 = 33$. ScalaR's aggregation calculations produce the following query:

```
select avg(latitude), avg(longitude)
from (select latitude, longitude
      from quake)
regrid 32, 33
```

where ScalaR uses SciDB's `regrid` statement to reduce the result. This query tells SciDB to divide quake into subarrays with dimensions 32 by 33 alongx and y. The subarrays are summarized by taking the average of the latitude coordinates and the average of the longitude coordinates within in each subarray. The resulting array has 2479 non-empty cells, and Figure 2b shows the resulting plot. Note that most cells are empty, as most earthquakes occur in specific concentrated areas. `quake`'s dimensions represent latitude and longitude ranges. With aggregation, ScalaR was able to produce a visualization that is very similar to the original, with less than one third the number of points.

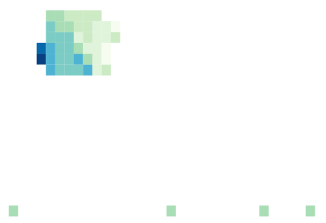


Figure 4. Overview visualization of the `ndvi_points` array

Now suppose we ask ScalaR to perform sampling over `quake`, using the number of points produced using aggregation as the threshold. ScalaR computes the sampling rate to be the desired resolution divided by the size of the original data set:

$$p = \frac{n}{|\text{quake}|} = \frac{2479}{7576}$$

In this example, p is 0.327.Sampling to reduce the resolution produces the following query:

```
select latitude, longitude
from bernoulli(
      (select latitude, longitude
       from quake),
      0.327,
      1)
```

where the original query is wrapped in a SciDB `bernoulli` statement, and the default seed is 1. This query tells SciDB to randomly choose points from quake, where each point is chosen with a probability of 0.327.In this case, sampling results in 2481 data points, which ScalaR prunes to 2479 to satisfy the threshold conditions by randomly choosing 2 points to remove from the reduced result. Figure 2c shows a plot of the query result. Like aggregation, sampling produces a visualization very similar to the original visualization with considerably less data.

Now that the user has identified the regions with the most earthquakes, she can filter the data in favor of these regions. This results in the following query to retrieve points in regions 2 and 3 (shown in Figure 2d):

```
select latitude, longitude
from quake
where lat > 20 and
(lon < -100 or lon > 170)
```

As shown in Figure 3, she can then zoom into regions 2 and 3 to see the distribution of earthquakes in more detail.

B. *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 California, 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 NDVI data over the southern California coast. The user first writes a query to retrieve all data from `ndvi_points`:

```
select ndvi from ndvi_points.
```

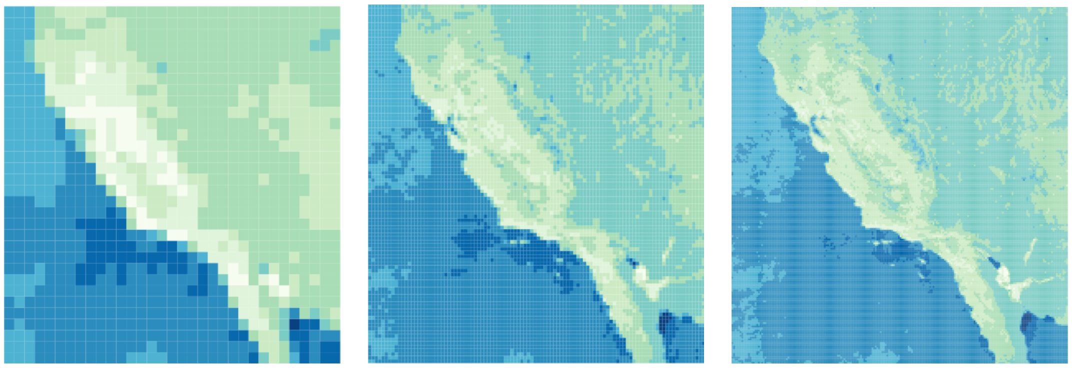

(a) 1,000 points resolution (b) 10,000 points resolution (c) 40,000 points resolution

Figure 5. Zoom on the California region of the `ndvi_points` array at 1,000, 10,000, and 40,000 points resolution

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 (see Section II). This visualization clearly shows the array's sparseness, and reveals a dense area of data in the array.

Now the user zooms in on the dense portion of the array by highlighting the area with a selection box and using the "zoom-in" button. 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.

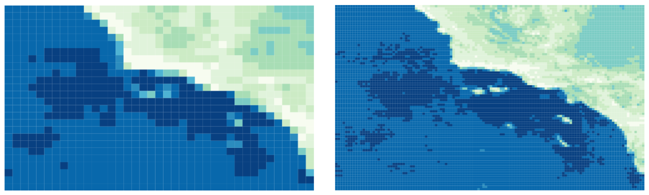

(a) 1,000 points resolution (b) 10,000 points resolution

Figure 6. Zoom on LA area at 1,000 and 10,000 points resolution

Resolution	Aggregation Runtime (s)	Sampling Runtime (s)
1,000	89.55	1.95
10,000	87.22	1.94
100,000	88.71	24.52
1,000,000	98.58	133.68
10,000,000	132.32	176.58
100,000,000	1247.78	186.90
1,000,000,000	3692.02	296.83
Baseline	210.64	

Table I
RAW RUNTIME RESULTS IN SECONDS FOR AGGREGATION AND SAMPLING QUERIES OVER THE `NDSI1` ARRAY, WITH VARIOUS RESOLUTION VALUES. EXECUTION TIME FOR A FULL SCAN OVER `NDSI1` IS PROVIDED FOR REFERENCE, LABELED AS THE BASELINE.

V. PERFORMANCE

We used a 2-node SciDB cluster to run the following experiments. Each node had 50GB of RAM, 32 cores, and 10.8TB of disk space. SciDB was limited to using at most 75% of the available memory per node (as recommended by the SciDB User's Guide [16]), but the operating system still had access to all available memory. We measured the execution times of aggregation and sampling queries over a single SciDB array containing Normalized Difference Snow Index calculations (NDSI) for the entire world, which where computed over roughly one week of NASA MODIS data. The normalized difference snow index measures the amount of snow cover on the earth at a given latitude-longitude coordinate. For the rest of this section, we will refer to this array as `ndsi1`. The `ndsi1` array was roughly 209GB on disk when stored directly inside SciDB, and 85GB when stored as a compressed SciDB binary file. `ndsi1` was a sparse array containing over 2.7 billion data points, stored across 673,380 different SciDB chunks. We varied

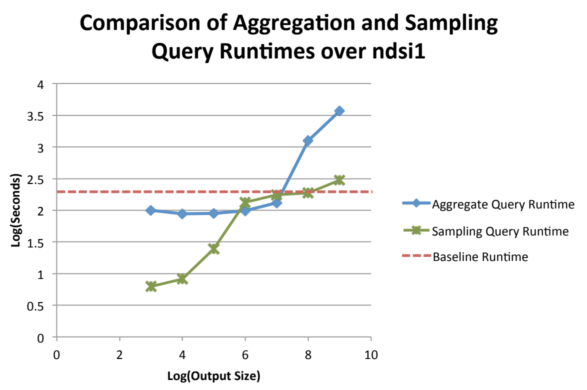


Figure 7. A comparison of aggregation and sampling on the `ndsi1` array with various data thresholds

the resolution threshold (*i.e.* maximum output size) from one thousand to one billion data points, and measured the runtime of the resulting SciDB aggregation and sampling queries dispatched by ScalaR. As a baseline for comparison, we also measured the execution time for a full scan of the `ndsi1` array (*i.e.* "`SELECT * FROM ndsi1`").

We present runtime results in Table I, and a log-scale comparison of aggregation and sampling in Figure 7. Our preliminary results show that basic aggregation and sampling are effective in reducing output size and execution time for most recorded output sizes. We see in Figure 7 that reducing the resolution of `ndsi1` via sampling either significantly improves performance or is on par with the baseline. Aggregation performs better than or as fast as the baseline for most resolution sizes, but slower than sampling. We also see that performance plummets at the highest resolution sizes. Aggregation's slower performance is due in part to the fact that the `ndsi1` array is sparse. Aggregation computes over logical array ranges, making it less efficient when reducing sparse arrays. In addition, as the resolution increases, aggregation performs even more operations per SciDB chunk. Chunks are SciDB's unit of storage on disk. At resolutions of 100 million and one billion data points, aggregation is executing hundreds or more operations per chunk, causing aggregation's poor performance.

Note that our simple reduction algorithms require reading virtually the entire data set, limiting their performance. We plan to implement more efficient reduction techniques in the future, and compare their performance to our basic algorithms.

VI. Conclusions and Future Work

We presented the design and implementation of ScalaR, an information visualization system that dynamically performs resolution reduction to improve query execution performance of clusters running a distributed DBMS. ScalaR uses aggregation, filtering and/or sampling operations to downsize query results as necessary to reduce completion time while still producing visualizations close in accuracy to the original result. We presented preliminary performance results for 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 use machine learning techniques over existing visualizations found on the web to learn how to choose appropriate visualization types for user query results automatically. 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] "Tableau software," http://www.tableausoftware.com/, May 2012.

[2] "Tibco spotfire," http://spotfire.tibco.com/, May 2012.

[3] H. Vo *et al.*, "Parallel visualization on large clusters using mapreduce," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, 2011, pp. 81–88.

[4] "Hadoop," http://hadoop.apache.org/.

[5] D. Jerding and J. Stasko, "The information mural: a technique for displaying and navigating large information spaces," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 4, no. 3, pp. 257–271, 1998.

[6] N. Elmqvist and J. Fekete, "Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines," *IEEE Trans on Visualization and Computer Graphics*, vol. 16, no. 3, pp. 439–454, 2010.

[7] S. Chaudhuri and U. Dayal, "An overview of data warehousing and olap technology," *SIGMOD Rec.*, vol. 26, no. 1, pp. 65–74, Mar. 1997.

[8] J. M. Hellerstein *et al.*, "Online aggregation," *SIGMOD Rec.*, vol. 26, no. 2, pp. 171–182, Jun. 1997.

[9] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," *SIGMOD Rec.*, vol. 28, no. 2, pp. 287–298, Jun. 1999.

[10] J. M. Hellerstein *et al.*, "Interactive data analysis: The control project," *Computer*, vol. 32, no. 8, pp. 51–59, Aug. 1999.

[11] D. Fisher *et al.*, "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*, ser. CHI '12. New York, NY, USA: ACM, 2012, pp. 1673–1682.

[12] S. Agarwal *et al.*, "Blinkdb: queries with bounded errors and bounded response times on very large data." New York, NY, USA: ACM, 2013, pp. 29–42.

[13] M. Bostock *et al.*, "D3: Data-driven documents," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[14] "Google maps api," https://developers.google.com/maps/, May 2012.

[15] P. Cudre-Mauroux *et al.*, "A demonstration of scidb: a science-oriented dbms," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1534–1537, Aug. 2009.

[16] "Scidb user's guide (version 13.3)," 2013. [Online]. Available: www.scidb.org