Interactive Visualization of Big Data Leveraging Databases for Scalable Computation

by

Leilani Marie Battle

B.S., University of Washington (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author .....
Department of Electrical Engineering and Computer Science
August 23, 2013

Certified by .....
Michael R. Stonebraker
Adjunct Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by .....
Samuel R. Madden
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by.....
Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Interactive Visualization of Big Data Leveraging Databases for Scalable Computation

by

Leilani Marie Battle

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 2013, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

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 thesis presents the design and implementation of ScalaR, and shows results for two example applications, visualizing earthquake records and satellite imagery data, stored in SciDB as the back-end DBMS.

Thesis Supervisor: Michael R. Stonebraker
Title: Adjunct Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Samuel R. Madden
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I want to first thank my advisors Sam and Mike for their keen insight and constant encouragement, and for helping me learn to believe in myself. I would like to thank Remco Chang for his patient mentorship and valuable feedback throughout the many iterations of this work. I would also like to thank the Paradigm4 team for their extensive support and insight as I was worked with the SciDB system. Last but not least, I would like to thank my husband Ray, family, and friends for the endless support they provided me throughout these first two years at MIT.

Contents

List of Figures

List of Tables

Chapter 1

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 [8], Spotfire [9], R and Matlab, are actively used in conjunction with database management systems. Furthermore, Bronson *et. al.* [30] show that distributed data management and analysis systems like Hadoop [2] 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. 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 improve performance on massive data sets, many large-scale visualization systems rely on fitting the entire data set within memory, tying data analytics and visualization directly to the management of the data. However, without a modularized approach to designing large-scale visualization systems, future systems will have to act as their own database management systems. 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 named ScalaR that leverages the computational power of modern database management systems for back-end analytics and execution. ScalaR decouples the task of visualizing data from the analysis and management of the data by inserting a middle layer of software to mediate between the front-end visualizer and back-end DBMS. ScalaR is back-end agnostic in design, as its only requirements are that the back-end must support a query API and provide access to metadata in the form of query plans (see Section 3.1 for more information about query plans). ScalaR relies on query plan estimates computed by the DBMS to perform resolution reduction, or to summarize massive query result sets on the fly. We provide more details on resolution reduction below.

There are many existing systems and techniques for reducing or summarizing the data the front-end has to visualize. One prevalent approach is using OLAP [13], where the data is preprocessed in advance, producing aggregated data cubes. Liu *et. al.* [24] use this technique in conjunction with custom WebGL optimizations in the front-end to visualize billions of data points. Another example of limiting the amount of data executed over is by iterative, approximate query execution, proposed initially by Hellerstein *et. al.* [21, 19, 20], and specifically within the context of data visualization by Fisher *et. al.* [17]. The database starts by sampling a small fraction of the data set, which produces an initial result with low accuracy, and continues to improve in accuracy until it reaches the user's desired resolution.

We provide a more general approach to reducing the amount of data executed over by performing resolution reduction. The front-end visualization system specifies a limit in advance on the amount of data the back-end DBMS can return. This data limit can be driven by various performance factors, such as resource limitations of the front-end visualization system. We insert a middle layer of software between the front-end visualization system and DBMS that determines when a query will violate the imposed data limit and delegates to the DBMS how to reduce the result as necessary. 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 thesis, we make the following contributions:

- We present a modularized architecture for a scalable information visualization sys-

tem that is completely agnostic to the underlying data management back-end

- We present an approach for automatic query result reduction using query plans for limit estimation and leveraging native database operations to reduce query results directly in the database.
- we present 2 motivating examples for ScalaR, visualizing recorded earthquakes and NASA MODIS satellite imagery
- We present preliminary performance results for using ScalaR to visualize NASA MODIS satellite imagery data.

1.1 Related Work

The design and implementation of scalable, parallelized visualization systems is a well-researched area within the scientific visualization and high performance computing (HPC) communities. For example, VTK [25] is an open-source programming toolkit for 3D graphics, image processing and interactive visualization. Several popular large-scale visualization systems have been built using VTK, including ParaView [27] and VisIt [14]. Streaming techniques for progressive processing of data is also a prevalent paradigm for visualizing very large data sets. One example of such is the ViSUS project [6]. These systems do not require a scalable back-end to visualize massive data sets as they perform their own data management and parallelization directly in the visualizer. However, the downside of these systems is their limited adoptability within research communities. They are highly specialized, often designed to run directly on supercomputers, and are intended for more sophisticated 3D visualization and simulations. Domain scientists may find these systems challenging to learn and integrate into existing workflows, and may not have access to supercomputers to run these systems.

Many visualization systems have also been designed and implemented within the database community. Ahlberg *et. al* presented dynamic queries [11] to make visual database exploration more intuitive. Dynamic querying allows the user to quickly filter out unrelated data by tying data attributes directly to elements in the user interface, such as sliders and buttons.

Manipulation of the interface automatically generates filtering operations on the underlying data. Popular systems using this technique for database exploration include Polaris [28] (now Tableau [8]) and Spotfire [9]. There are also many other systems designed for visual data exploration, such as VisDB [23] and Tioga [29]. These systems make visually exploring databases straight-forward and intuitive, and several of these systems scale up to millions of data points or more. However, most of these systems do not scale to accommodate massive data sets, as they do not use front-end or back-end reductions to efficiently manage query results. In addition, these systems often connect directly to a DBMS, leaving them susceptible to a huge influx of query results from the back-end DBMS, and slow performance on user queries over massive data sets.

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.* [22] 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.* [16] 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 [13]. 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.* [21, 19, 20] 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.* [18] 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 [10] 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.

Chapter 2

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 2-1). They are described in detail below.

2.1 Web Front-End

We implemented a web-based front end, using the D3.js [12] Javascript library to draw the visualizations. ScalaR supports scatterplots, line charts, histograms, map plots and heat maps. The Google Maps API [7] 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, or 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.

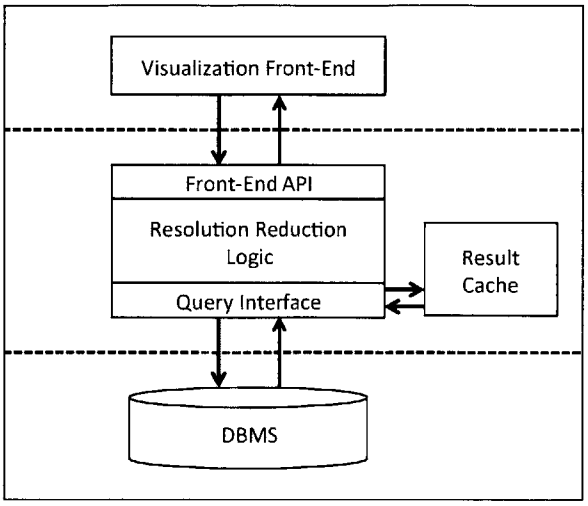


Figure 2-1: ScalaR system architecture.

2.2 Intermediate Layer

The intermediate layer is the major component of the ScalaR system. 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.

The intermediate layer has 4 separate subcomponents:

- A front-end API the visualizer uses to communicate with the intermediate layer
- A module of functions for determining if resolution reduction is necessary and for computing the necessary reductions for user queries
- A back-end interface for communicating with the DBMS and translating resolution reduction into queries
- A result cache used to store previously fetched data across all users

The front-end visualizer sends user-defined queries and query result size limits to the intermediate layer through the front-end api. Before dispatching user queries to the DBMS, the intermediate layer retrieves the proposed query plan from the DBMS and parses it to identify the expected size of the result. The expected result size is compared with the provided front-end restrictions to determine whether to insert a resolution reduction operation into the original query. See Section 3.2 for a list of resolution reduction operations. If the

result of the user's query needs to be reduced, the intermediate layer computes the reduction parameters for the new reduced query, and dispatches the new query to the DBMS. Results retrieved from the DBMS are stored in the result cache. If the result cache is full, a straight-forward policy such as LRU is used for eviction.

2.3 DBMS

ScalaR is database agnostic in design, but 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 stored in each cell in the matrix (see Section 4.1 for examples of SciDB queries).

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.

Chapter 3

Resolution Reduction

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

3.1 Retrieving Metadata From the Query Optimizer

Almost all DBMSs 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 tuples to be produced by the query.

Modern DBMSs 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 thesis. 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 DBMSs 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. Suppose you have a table called `earthquake` stored in a PostgreSQL database, and you want to know how PostgreSQL will execute a query retrieving every tuple from this table:

```
SELECT * FROM earthquake;
```

To generate a query plan for this query, you would execute the following in PostgreSQL:

```
EXPLAIN SELECT * FROM earthquake;
```

This will produce the following query plan:

```
                          QUERY PLAN
---------------------------------------------------------------
 Seq Scan on earthquake (cost=0.00..16.90 rows=790 width=43)
(1 row)
```

The above query plan explains what operations PostgreSQL will use to execute the query, the cost of executing the query, the estimated number of tuples in the result, and the estimated width of the tuples.

"`Seq Scan on earthquake`" tells us that PostgreSQL plans to perform a sequential scan over the data. PostgreSQL estimates that the query will produce 790 rows, given by "`rows=790`". PostgreSQL also exposes an internal cost value associated with executing this query plan, given by "`cost=0.00..16.90`". This value is used for internal cost optimization purposes, is not directly comparable across different query operations, and has no direct units of measurement. The estimated width of the resulting tuples is 43, given by "`width=43`". Users can specify more options when executing the `EXPLAIN` command to retrieve additional query plan information. See the PostgreSQL reference manuals [4] for more information on retrieving query plan information from PostgreSQL.

Now suppose we have the same data set stored in a SciDB array, 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, you would execute the following in SciDB:

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

This produces the following query plan:

```
[("[pPlan]:
>[pNode] physicalScan agg 0 ddl 0 tile 1 children 0
  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]
  props sgm 1 sgo 1
  distr roro
  bound start {1, 1} end {6381, 6543} density 1
  cells 41750883 chunks 1 est_bytes 7.97442e+09
")]
```

"`physicalScan`" tells us that SciDB will be performing a sequential scan of the data. The whole schema of the resulting array is provided beginning on line three 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. There are 41,750,883 cells in the array, given by "`cells 41750883`" on the last line of the query plan. The SciDB stores the array in a single chunk, given by "`chunks 1`" on the same line. Chunks are SciDB's unit of storage on disk. 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.

3.2 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 when scaling to massive data sets. Also, these systems lack effective tools for automatically aggregating results to avoid over-plotting. 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). Aggregation divides the array in logical space into evenly-sized sub-arrays, and performs an aggregate operation on the sub-arrays, such as computing the average of all points in each sub-array. When the data set is dense, the logical representation of the data matches the physical storage layout of the data, and aggregation significantly reduces the output size by grouping points by proximity. However, when the data is sparse, aggregation is not as effective. The logical range of each sub-array may be much larger than the physical space required to store the underlying data due to a high percentage of null values, causing aggregation to perform more work over fewer data points. Sampling in contrast is much more effective for sparse data sets, as it operates over only the non-null cells of the array, and empty logical ranges are totally ignored. When the user knows in advance what predicates to use to filter out non-relevant data, direct filtering is also an option.

Aggregation and sampling take an implicit parameter n that represents the maximum resolution, or the maximum total data points, the reduced query can return. Filtering takes a set of predicates for removing non-relevant data. The techniques are as follows:

Aggregation: The data is grouped into n sub-matrices, and aggregate operations are computed over the sub-matrices. Aggregate operations include: sum, average, max and min.

Sampling: Given a probability value p, effectively flip a coin biased with p for each data point in the original data set to determine what fraction of the data to return, where $p * |\text{data}| = n$. Most databases already support this operation.

Filtering: Given a set of desired characteristics over the data, only retrieve the elements that meet these characteristics. These characteristics are translated into filter (*i.e.* `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.

3.3 Analyzing SciDB Query Plans for Resolution Reduction

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 3.1. 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 3.2 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.

3.3.1 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 cells in the sub-array.

SciDB already supports our aggregation reduction technique through its default `regrid` function. However, SciDB uses slightly different syntax, and does not take the number of desired sub-arrays n as input. We must instead pass as input to SciDB the 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.

Algorithm 1 demonstrates how ScalaR performs aggregation over a d-dimensional array A. 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.

Algorithm 1 Resolution Reduction by Aggregation

Require: array A, resolution n, aggregate operation `op`, array attribute a_j
Ensure: $|A'| \leq n$

```
1: procedure AGGREGATE(A,n,op,a_j)
2:     s_1,s_2,...,s_d ← GetSubArrayDims(A,n)
3:     q = "regrid(A,s_1,s_2,...,s_d,op(a_j))"
4:     A' = db.execute(q)
5:     return A'
6: end procedure
7: procedure GETSUBARRAYDIMS(A,n)
8:     n_d = ⌊∛n⌋
9:     for all dimensions i do
10:        s_i = max(1, ⌈width_i(A)/n_d⌉)
11:    end for
12:    return s_1,s_2,...,s_d
13: end procedure
```

3.3.2 Sampling

When a user requests that data be sampled to reduce the resolution, ScalaR returns a random sample of the result. Most DBMSs already provide their own random sampling operations. SciDB's `bernoulli` function performs random 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.

Algorithm 2 demonstrates how ScalaR performs sampling over a d-dimensional array A. 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'.

Algorithm 2 Resolution Reduction by Sampling

Require: array A, resolution n
Ensure: $|A'| \leq n$

```
1:  procedure SAMPLE(A,n)
2:      p = n/|A|
3:      q = "bernoulli(A,p,SEED)"
4:      A' = db.execute(q)
5:      if |A'| > n then
6:          t = |A'| − n
7:          while t > 0 do
8:              remove a random element x from A'
9:              t = t − 1
10:         end while
11:     end if
12:     return A'
13: end procedure
```

3.3.3 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 [11], 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.

Chapter 4

Motivating Examples

We now present two scenarios for using ScalaR that demonstrate how our system solves the issues presented in Chapter 1.

4.1 Earthquake Data

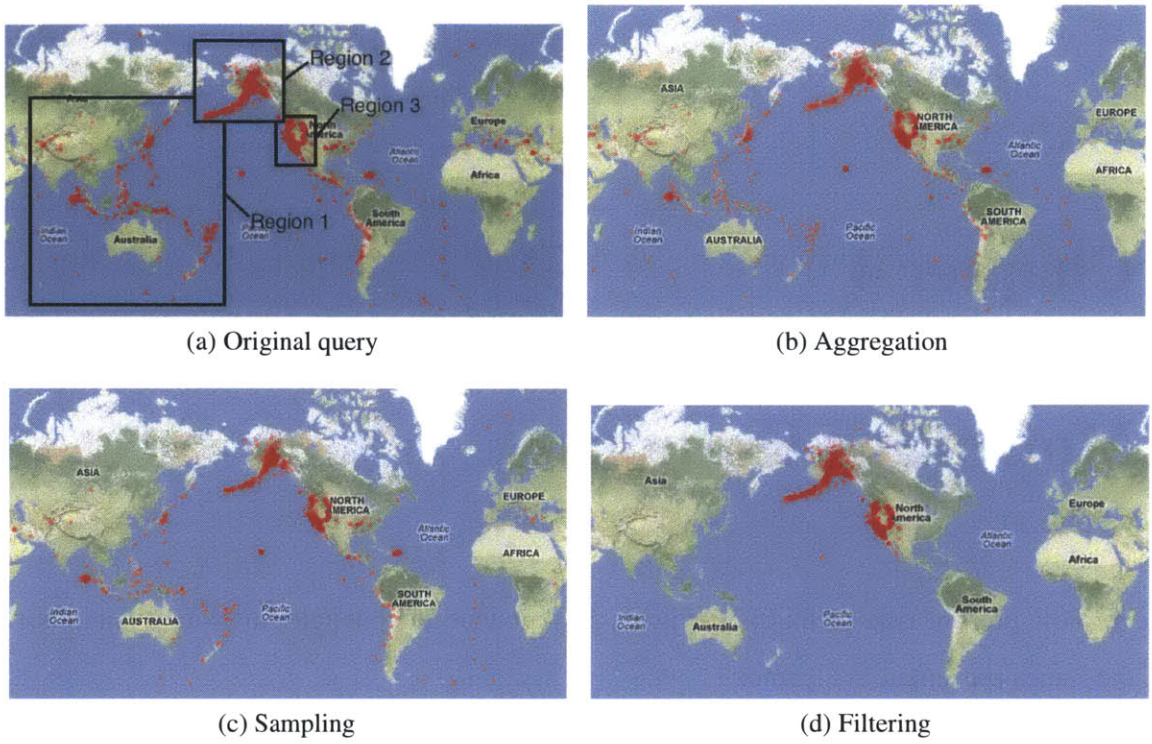
Suppose a user of the ScalaR system wants to plot earthquake data to see the distribution of earthquakes around the world. She inputs the following query, and requests a map plot of the results:

```
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, latitude, longitude)[x,y]
```

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.



(a) Original query

(b) Aggregation

(c) Sampling

(d) Filtering

Figure 4-1: Map plots for a query manipulated using several resolution reduction techniques.

The user picks 3 major regions of interest in this plot, identified by the three boxes drawn in Figure 4-1a. 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 4-1a shows a classic over-plotting problem with map visualizations, where each recorded earthquake is represented as a red dot. Region 1 in Figure 4-1a 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 2,423 points (over 30%), and region 3 has 4,081 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 use aggregation to divide the logical range of `quake` into no more than 40,000 sub-arrays. ScalaR first takes the dth root of n to compute the number of subarrays along every dimension n_d (see Section 3.3 for more information):

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

where d is the number of dimensions (2) and n is our desired resolution (40,000). n_d is 200 in 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:

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

where 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, 3
```

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 along `x` 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 2,479 non-empty cells, and Figure 4-1b shows the resulting plot. Note that most of the cells are empty because most of the earth did not have earthquakes occur during this time frame. `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. However, the aggregated plot is still misleading, as we are aggregating the original longitude and latitude coordinates of the earthquakes and plotting new points. The locations displayed in Figure 4-1b are not real. In addition, outliers are lost in the map plot when aggregation combines them with neighboring points.

Now suppose we instead ask ScalaR to perform sampling over `quake`, using the num-

ber of points produced using aggregation as the threshold. ScalaR would compute the sampling rate to be the desired resolution divided by the size of the original data set:

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

where in this example, $p = 0.327$. 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 2,481 data points, which ScalaR prunes to 2,479 to satisfy the threshold conditions by randomly choosing 2 points to remove from the reduced result. Figure 4-1c 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 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 (shown in Figure 4-1d):

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

As shown in Figure 4-2, she can then zoom into regions 2 and 3 to see the distribution of earthquakes in more detail.

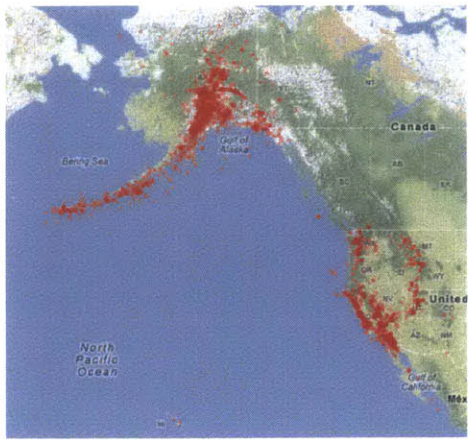


Figure 4-2: Zoom on regions 2 and 3 over filtered query results.

4.2 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, 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 latitude and longitude range of the `ndvi_points` array covers the entire earth, but most of the array is empty. Only NDVI calculations over the California and Northern Mexico area are stored. The NDVI calculations were visualized as heat maps, 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 heat map 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.

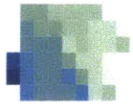
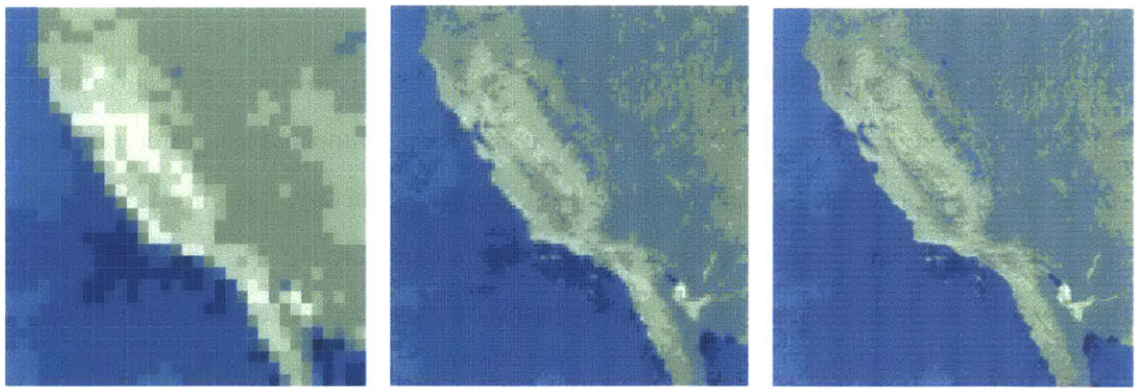


Figure 4-3: Overview visualization of the `ndvi_points` array

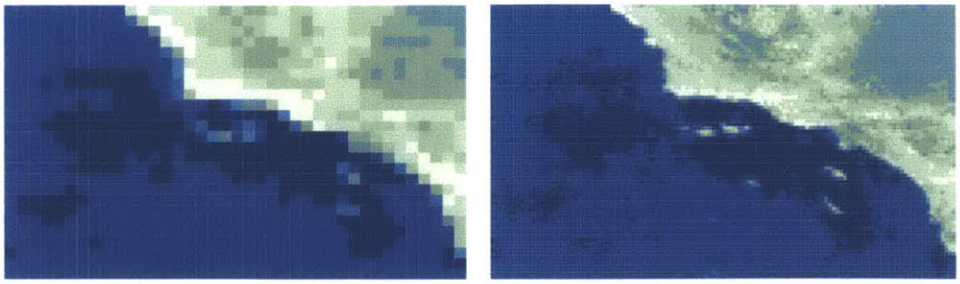
Using aggregation, ScalaR produces an initial visualization at a resolution of about 1,000 points, shown in Figure 4-3. Resolution refers to the size of the query results being drawn, so Figure 4-3 shows the result of reducing the data down to a 31 by 31 matrix (see Chapter 2). The large amount of whitespace in the visualization clearly shows the sparseness of `ndvi_points`, and reveals a single dense area of data in the array, which we know to be the California region.



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

Figure 4-4: Zoom on the California region of the `ndvi_points` array at 1,000, 10,000, and 40,000 points resolution

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 1,000 points is shown in Figure 4-4a. The general shape of the western coast of California/Northern Mexico is apparent, but the user may want the image to be clearer.



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

Figure 4-5: Zoom on LA area at 1,000 and 10,000 points resolution

Figures 4-4b and 4-4c show the results of increasing the resolution to 10,000 and 40,000 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 4-5.

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.

Chapter 5

Performance

In this chapter, we present performance results for aggregation and sampling reduction queries over two arrays containing snow cover measurements computed from NASA MODIS satellite imagery data.

5.1 Experimental Setup

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 [26]), 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 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, we also measured the runtime for performing a full scan of the ndsi1 array (*i.e.* " SELECT * FROM ndsi1").

We also computed a much smaller, aggregated version of the ndsi1 array that contained roughly six million points, which we will refer to as ndsi2. ndsi2 was a dense array, where roughly 93% of cells were non-empty. The data in ndsi2 was stored across 648 SciDB chunks. We measured the runtime of various aggregation and sampling queries for ndsi2, and also compared the quality of the visualizations produced for each aggregation and sampling query. We used Processing [5] instead of D3 to draw the following visualizations.

5.2 Results

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 5.1: 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 the ndsi1 array is provided at the bottom of the table for reference, labeled as the baseline.

The timing results for our general performance test on array ndsi1 are provided in Figures 5-1 and 5-2 and Table 5.1. For reference, the time required to execute a full scan (*i.e.* SELECT * query) over the ndsi1 array is provided at the bottom of Table 5.1. We will refer to the execution time for a full scan over ndsi1 as our baseline for execution time over ndsi1. We varied the resolution of the output by seven different orders of magnitude, from one thousand to one billion data points, and recorded query execution times for aggregation sampling queries. Figures 5-1 and 5-2 reveal several interesting points:

- Aggregation and sampling have very different performance tradeoffs in SciDB

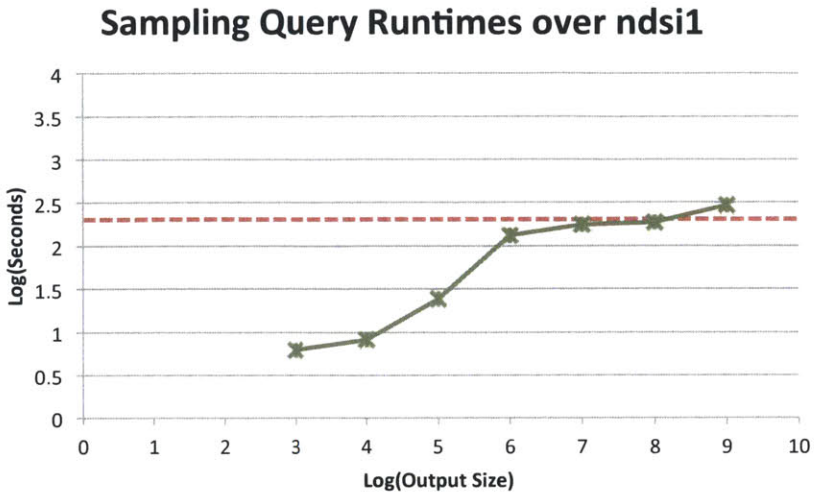


Figure 5-1: Runtime results for aggregation and sampling reduction queries on the `ndsi1` array with various data thresholds. The baseline is denoted as a dashed line.

- Sampling has minimal overhead in SciDB, and is in general significantly faster than aggregation
- Aggregation is impractical as the output size approaches the size of the original data set.

The following sections explore in detail ScalaR's aggregation and sampling performance over `ndsi1`, and how the above factors change for the denser data set `ndsi2`. We also visualize the results from each aggregation and sampling reduction over `ndsi2`, and compare the quality of each set of visualizations.

5.2.1 Sampling Performance

As demonstrated in Figure 5-1, sampling in general is a fairly low-overhead operation in SciDB. For very small output sizes, sampling is extremely fast, finishing in tens of seconds. There is a small cost associated with sampling which involves deciding what data points to include in the sample, causing sampling to be more expensive than a full scan of the array as the output size approaches the size of the original data set. From one thousand to one million data points, we see a non-linear increase in runtime. This is due to the fact that output size is increasing exponentially. `ndsi1` is stored as 673,380 SciDB chunks. While

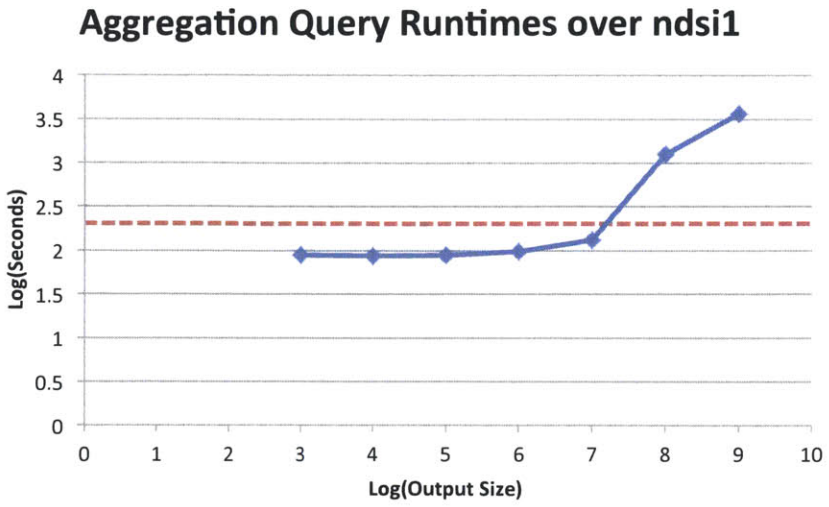


Figure 5-2: Runtime results for aggregation reduction queries on the `ndsi1` array with various data thresholds. The baseline is denoted as a dashed line.

the output size is small (*i.e.* less than 673,380), the output size is determining the number of chunks that need to be read to retrieve the final data points. Thus as the output size increases in magnitude, the number of chunks retrieved and executed over also increases in magnitude. Runtime then tapers off and becomes linear for output sizes of one million data points and above, because a high percentage of SciDB chunks are consistently being retrieved for each sample.

5.2.2 Aggregation Performance

We see in in Figure 5-2 that aggregation performs very differently than sampling. For output limited to a size of one thousand to ten million data points, aggregation is much faster than the baseline, producing results in half the time. Aggregation is also very consistent over this range of output sizes, producing results within 88 to 98 seconds. There is a slight decrease in runtime at an output size of ten thousand data points. This is likely due to 2 separate factors. First, while the output size is small, aggregation reduction queries (SciDB `regrid` operations here) are approximating general aggregation queries in SciDB, which operate over the entire data set. As output size increases to ten thousand data points, SciDB has to combine fewer intermediate results, making aggregation slightly faster to compute.

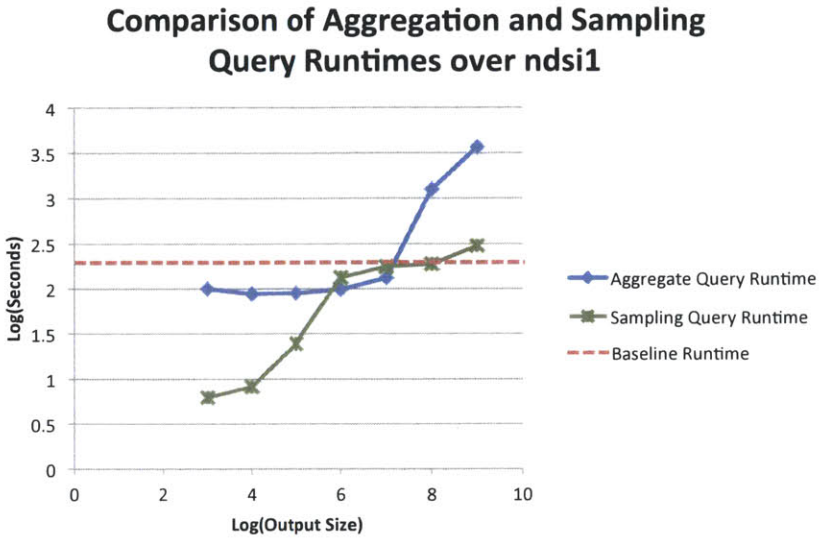


Figure 5-3: A comparison of aggregation and sampling on the `ndsi1` array with various data thresholds

Second, SciDB is able to process the smaller query results in dense array format, as it is highly likely that each cell of the result will be non-empty. However, as the output sizes increase beyond ten thousand data points, SciDB switches to sparse array format, which is slower. Larger output sizes result in sparse arrays because the original array is sparse.

However, we see that aggregation is considerably slower than the baseline for output sizes approaching the size of the original data set. This is mainly due to the fact the SciDB `regrid` operation is computation-intensive. From one thousand to one hundred thousand data points, SciDB is computing one aggregate operation over multiple chunks, which it can do efficiently as chunk retrieval is very fast. At an output size of one million data points, SciDB is computing roughly 1.4 aggregate operations per chunk, which only increases the runtime slightly, as the number of computations is still close in magnitude to the number of chunks. Aggregation runtime quickly skyrockets as the number of aggregate operations per chunk continues to increase in magnitude.

5.2.3 Comparing Aggregation and Sampling

Figure 5-3 provides a direct comparison of aggregation and sampling over the `ndsi1` array for a variety of output sizes. We see that sampling appears to be the straightforward choice

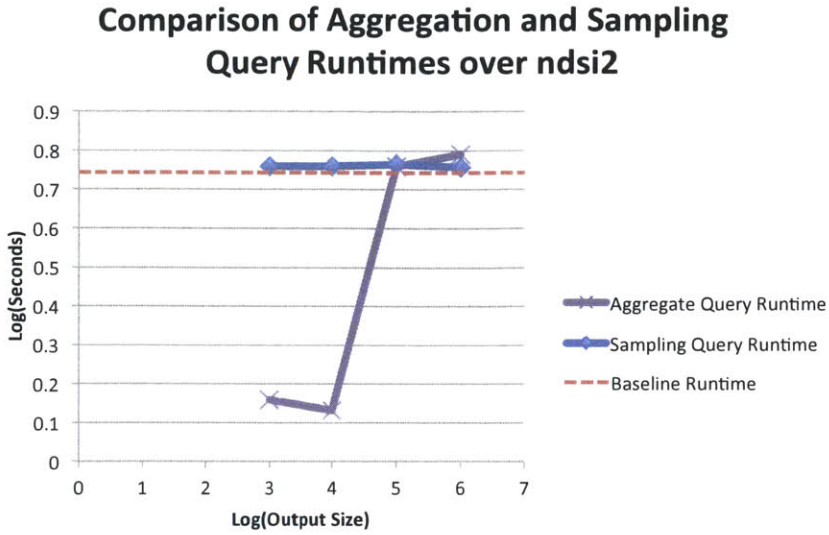


Figure 5-4: A comparison of aggregation and sampling on the `ndsi2` array with various data thresholds

for nearly all output sizes over `ndsi1`, as it is always faster or close in runtime to the baseline. However, producing results efficiently over a wide variety of array densities, and capturing the significant visual features of the original data set are equally important goals in ScalaR. In this section we present our results for repeating the above experiment with the small dense array `ndsi2`, and visualize the resulting output for both aggregation and sampling.

Resolution	Aggregation Runtime (s)	Sampling Runtime (s)
1,000	1.44	5.77
10,000	1.35	5.76
100,000	5.75	5.83
1,000,000	6.19	5.73
Baseline	5.68	

Table 5.2: Raw runtime results in seconds for aggregation and sampling queries over the `ndsi2` array, with various resolution values. Execution time for a full scan over the `ndsi2` array is provided at the bottom of the table for reference, labeled as the baseline.

Table 5.2 reports our runtime results for aggregation and sampling over the `ndsi2` array. The amount of time required to perform a full scan of `ndsi2` is provided at the bottom of Table 5.2. We will refer to this measurement as our baseline for execution time over `ndsi2`. Figure 5-4 presents a log-scale comparison of aggregation and sampling

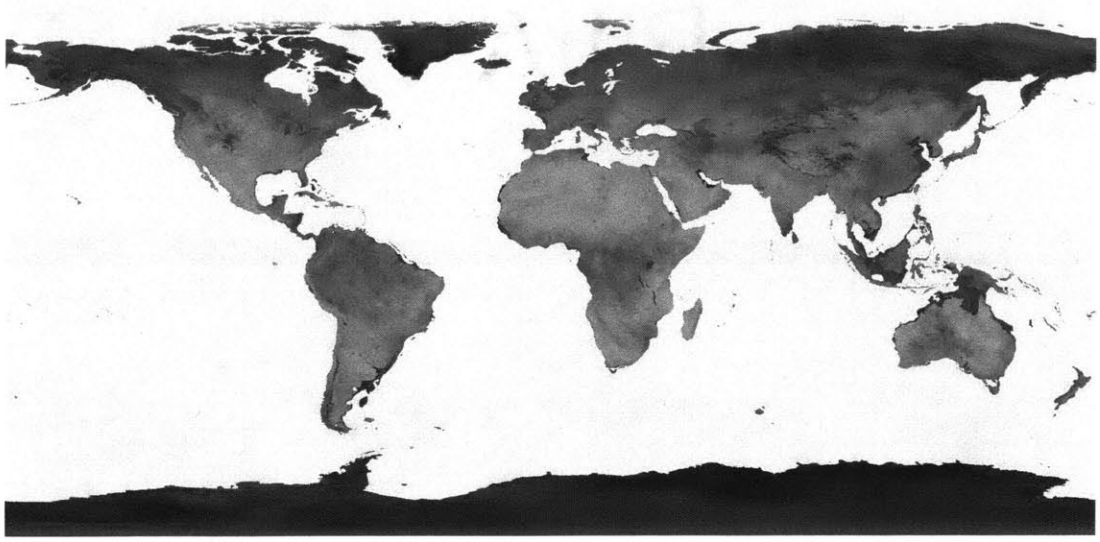
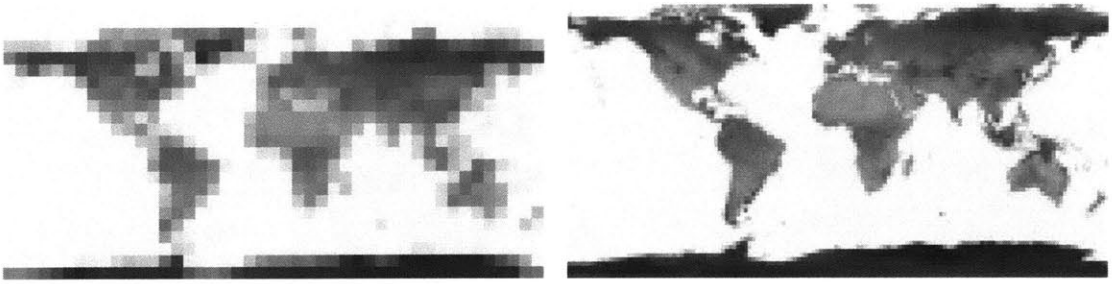


Figure 5-5: Greyscale visualization of the full `ndsi2` data set, where high NDSI values correspond to very dark areas in the visualization.

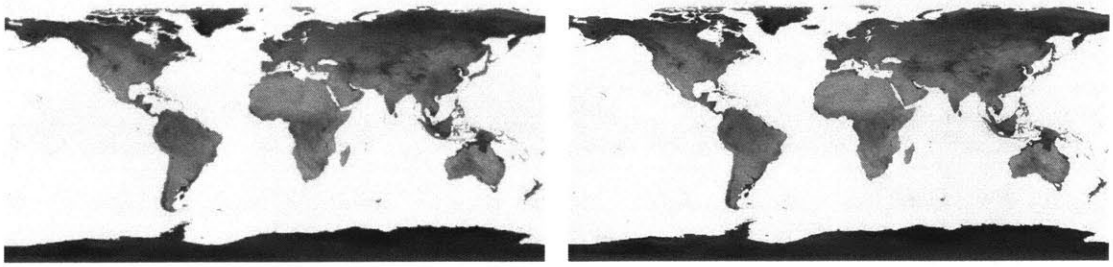
runtimes, where the baseline runtime is denoted with a dashed line. `ndsi2` is several orders of magnitude smaller than `ndsi1`, and all queries where executed in a matter of seconds. We see in Figure 5-4 that sampling does not outperform aggregation over `ndsi2` as it did over `ndsi1`. This is due to the fact that sampling is effective over sparse data sets, but more costly over dense data sets in SciDB. Thus sampling performance is worse over `ndsi2` than `ndsi1`. In contrast, SciDB can perform aggregations more efficiently over dense data sets, making aggregation faster over `ndsi2` than `ndsi1`. We also see that aggregation still suffers in performance when many aggregate operations are being executed per chunk, which happens here at output sizes of one hundred thousand and one million data points. Sampling shows only small increases in runtime with increases in output size due to a high percentage of chunks being retrieved, as there are only 612 chunks in `ndsi2`.

Visualization quality is also an important factor in determining which reduction technique is most appropriate for a given data set. To demonstrate visualization quality for our `ndsi2` reductions, we also provide the corresponding heat map visualizations of the NDSI results from each aggregation and sampling reduction. Figure 5-5 is the result of visualizing the entire `ndsi2` array in greyscale. High NDSI calculations (*i.e.* snow) corresponds to very dark areas in the visualization. Note that costal waters often cause false positives



(a) Aggregating NDSI data at 1,000 points resolution

(b) Aggregating NDSI data at 10,000 points resolution



(c) Aggregating NDSI data at 100,000 points resolution

(d) Aggregating NDSI data at 1,000,000 points resolution

Figure 5-6: Aggregating NDSI data to an output size of 1,000, 10,000, 100,000 and 1,000,000 points resolution

in the data set, so some coastal areas that do not have snow will appear dark in the visualization. This high-resolution image contains over 6 million data points, and will act as our visual baseline for output quality.

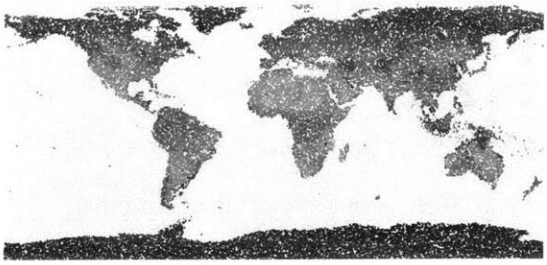
We visualized four different output sizes from one thousand to one million data points for both aggregation and sampling. The results are presented in Figure 5-6 and Figure 5-7. From these visualizations we see the following properties for heat maps:

- Aggregation is able to visualize the full range of the data set for all output sizes
- Sampling plots are very sparse and difficult to read at very small output sizes
- Both aggregation and sampling can provide effective visual approximations of the original data set with significantly less data than the original data set.

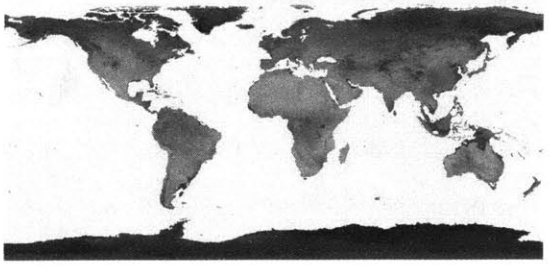
At just one thousand data points of output, we see in Figure 5-6a aggregation produces a visualization that appears to be a blurry map of the earth, capturing the visual cues the user needs to know that the data set is geographical in nature. In comparison, sampling

(a) Sampling NDSI data at 1,000 points resolution

(b) Sampling NDSI data at 10,000 points resolution



(c) Sampling NDSI data at 100,000 points resolution



(d) Sampling NDSI data at 1,000,000 points resolution

Figure 5-7: Sampling NDSI data to an output size of 1,000, 10,000, 100,000 and 1,000,000 points resolution

the equivalent number of points over the entire range of the data set produces a nearly empty visualization that without prior knowledge, would not appear to be geographical data. Given this new visual context, we see that very small output sizes do not produce useful visualizations when sampling. Thus making very small samples virtually useless for visual exploration of large data sets using heat maps.

Sampling also requires additional adjustment of the visualized points to best represent the data given the sparsity. For example, the visualized points are 20 pixels by 20 pixels in Figure 5-7a and Figure 5-7b, 10 pixels by 10 pixels in Figure 5-7c, and 5 pixels by 5 pixels in Figure 5-7d. With aggregation, we can directly compute the size of the aggregate regions to fill the visualization dimensions.

However, we see that with output sizes of ten thousand points or more, sampling and aggregation begin to produce useful approximations of the data set. For example, with an output size of ten thousand points (represented in Figure 5-7b and Figure 5-6b), we can start to identify regions known to receive high amounts of snowfall in the visualization, including Alaska, Northern Canada, Russia, and Antarctica. With an output size of one hundred

thousand points, less than 2% of the size of the original data set, aggregation and sampling produce visualizations very close in quality to our baseline in Figure 5-5. With one million data points, the aggregation and sampling visualizations are nearly indistinguishable from the original to the naked eye.

5.3 Discussion

We see in Figure 5-7b and Figure 5-6b that even summaries as small as 10,000 data points start to reveal significant features of the data set, which in this case is snow cover on the 7 continents. Computing small summaries take significantly less time, making them a powerful tool for fast visual exploration. Users can initially explore the data visualizing small summaries to quickly identify major features in the data set. After identifying regions of interest, they can can use ScalaR to visualize these specific areas to explore the data set in greater detail. We also see that the quality of the resulting visualization depends on visualization type, and aggregation is more effective than sampling for heat maps.

ScalaR's resolution reduction approach shows promise for summarizing massive data sets on the fly with varying levels of detail. However, section 5.2 shows that despite the use of SciDB's native operations to perform reductions, ScalaR's simple reduction algorithms are not yet executing at interactive speeds over a fairly small data set. This is due to the fact that ScalaR's current reduction approaches require executing over a significant portion of the original data set in order to produce the reduced versions. ScalaR's modularized design makes extending ScalaR with new reduction algorithms a simple and straightforward task. Future performance improvements include designing and implementing more efficient reduction algorithms for ScalaR and comparing their performance and visualization quality to the basic sampling and aggregation techniques explored here. See section 6.1 for a detailed description of proposed improvements to ScalaR's performance.

ScalaR's performance can also be improved through extending the back-end DBMS. For example, we can potentially make significant improvements in aggregation performance by pre-binning the dataset to better fit the aggregation model. Similar to BlinkDB's approach, building random samples over the data in advance for various sample sizes would

greatly improve sampling performance, and provide the user with additional information through error bounds. The addition of a main memory cache in SciDB could also greatly improve the performance of some queries, and warrants further investigation.

Chapter 6

Conclusion

We presented the design and implementation of ScalaR, an information visualization system that dynamically performs resolution reduction to make large-scale DBMS query results easier to manage by visualization systems. ScalaR's modularized architecture decouples the visualization of data from the management and analytics of the data, making it easy to incorporate existing database management systems to power the back-end. ScalaR uses DBMS query plans to estimate query result size and determine when to reduce query results. Reduction operations, such as aggregation or sampling, are inserted into user queries as necessary to enforce data limits imposed by the front-end visualization system. We presented two example applications of ScalaR, visualizing earthquake records and satellite imagery data stored in SciDB. We also presented performance results for aggregation and sampling reductions over satellite imagery data stored in SciDB.

6.1 Future Work

The ScalaR system provides the bare minimum functionality required for users to visualize their data. To improve the user experience with the ScalaR system, we have identified two key areas in which ScalaR can be improved:

- reducing the amount of required user intervention when building visualizations
- efficiently fetching query results from the back-end database to draw visualizations

The following sections describe in detail the issues associated with each area, and how we plan to extend ScalaR to address these issues.

6.1.1 Building Visualizations for Users Automatically

ScalaR's current design forces users to specify all components of the final visualization, including: what resolution reduction technique to use, the data limit to impose on the back-end, the x- and y-axes, the scaling factor, and coloring. ScalaR's intended use is for visual exploration, where the user may not have prior experience visualizing the data. Lack of experience with visualizing the underlying data makes it difficult for users to make specific visualization choices in advance, and can result in many iterations of trial and error as users search for a suitable way to reduce and visualize the data.

To help users quickly make better visualization-specific choices early in the exploration process, we are designing a predictive model for identifying the most relevant visualization types for a given data set. The predictive model will help ScalaR decide what visualization types, such as scatterplots or heat maps, are most appropriate or interesting for the given data set. Using this predicted list of visualizations, ScalaR will suggest possible visualizations over the data set that users can choose from to start exploring their data. A list of sample visualizations moves the work of deciding on a visualization type from the user to ScalaR. Pairing the predicted visualization types with simple statistics, such as correlation coefficients between pairs of attributes or the density and distribution of the array, ScalaR could also predict what aspects of the data should be represented in the visualization (attributes, dimensions, etc.). Given predicted visualization types and data mappings for the visualization types, ScalaR can start to suggest complete example visualizations to the user. ScalaR can then retrieve a small sample of data from the DBMS in advance to build thumbnails for the proposed visualizations. Providing real examples on the underlying data will allow users to decide on an initial visualization by looking at concrete examples, rather than relying solely on their intuition for what the visualizations should look like.

To train our predictive model for visualization types, we are creating a corpus of visualizations from the web. For each visualization in our corpus, we also have access to the

data set that was used to create the visualization. We will use the underlying data to learn what features potentially correspond to specific visualization types. The visualizations are collected from a wide variety of web sources, including the Many Eyes website [3], various ggplot2 examples [1], and the D3 image gallery [12]. Using visualizations from the web allows us to draw on a wide variety of sources, visualization types and data sets.

We also plan to make resolution reduction fully automatic by measuring the density of the underlying data set. For example, if the data is dense, aggregation may be the most effective way to reduce the data. If the data is sparse, sampling is likely the better choice. We will supplement the provided metadata from SciDB with additional density information that ScalaR can use to automatically choose a reduction type for the user. To decide on an appropriate data limit automatically for the user, ScalaR needs either more information about the visualization, such as the total number of pixels and visualization type, or feedback from the front-end when too much or too little data was returned. ScalaR can compute exactly how much data should be returned to the front-end without user intervention with direct information about the visualization. Given feedback from the front-end about the volume of data returned, ScalaR can learn an appropriate data limit automatically and dynamically adjust reduction estimates accordingly.

6.1.2 Prefetching Data

Since ScalaR depends on direct querying of the back-end database prior to visualization, ScalaR is susceptible to long wait times when retrieving query results. To help improve back-end response times when fetching data, ScalaR's current design incorporates caching of query results. However, ScalaR is currently only able to cache data after a user has already requested the results, exposing users to potentially long wait times for initial fetching of data tiles.

To better clarify the problem of prefetching data in ScalaR, we logically divide data into sets of non-overlapping, multi-dimensional sub-arrays called data tiles. Instead of visualizing arbitrary queries, the user uses a modified version of our map-style interface to view a few data tiles at a time. The data tile structure allows us to map the user's movements

to specific ranges in the underlying data as they use ScalaR's new map interface. Data tiles also provide us with discrete units in which we can prefetch data for the user.

We are developing several general prediction models for anticipating what data ranges the user will request next. The first model relies on predicting the user's physical trajectory through the data set. For example, knowing whether the user just zoomed in, or the direction the user is panning through the visualization can inform us of the possible paths the user is taking as she explores the data set. The second model attempts to predict relevant tiles by comparing tiles from the user's history; for example, the last two data tiles visited by the user. This model compares tiles by computing statistical similarities and differences between the data tiles, and using this information to find tiles with the same computed signature. This similarity model easily accommodates for switching between different algorithmic approaches for computing tile similarity. We can also extend the tile similarity model to support "search by example" by computing the k nearest neighbors for every tile using various comparison algorithms.

To make ScalaR's prediction scheme more robust, we plan to run our prediction algorithms in parallel, allotting a set amount of space to each model for storing prefetched data. As users explore data sets with this new architecture, we can monitor the success of each prediction scheme in real-time, and increase or decrease the allotted space for each model accordingly.

However, this poses new challenges in existing areas of ScalaR's architecture, primarily in the result cache. It is not yet clear what the best data eviction policy is when prefetched data and cached data from previous user queries are stored together in the result cache. This problem is made more complicated with multiple users, as not all users will have the same exploration behavior, and individuals may change their exploration strategy several times mid-session.

Bibliography

[1] ggplot2. ggplot2.org.

[2] Hadoop. http://hadoop.apache.org/.

[3] Many eyes. http://www-958.ibm.com/software/data/cognos/manyeyes/.

[4] Postgresql "explain" command documentation. http://www.postgresql.org/docs/9.1/static/sql-explain.html.

[5] Processing. www.processing.org.

[6] Visus. http://www.pascucci.org/visus/.

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

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

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

[10] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. pages 29–42, New York, NY, USA, 2013. ACM.

[11] Christopher Ahlberg, Christopher Williamson, and Ben 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.

[12] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3: Data-driven documents. *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[13] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[14] Hank Childs, Eric Brugger, Kathleen Bonnell, Jeremy Meredith, Mark Miller, Brad Whitlock, and Nelson Max. A contract based system for large data visualization. *Proc. IEEE Visualization 2005*, 2005.

[15] 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, August 2009.

[16] N. Elmqvist and J. Fekete. Hierarchical aggregation for information visualization: Overview, techniques, and design guidelines. *IEEE Trans on Visualization and Computer Graphics*, 16(3):439–454, 2010.

[17] D. Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 73–80, 2011.

[18] Danyel Fisher, Igor Popov, Steven Drucker, and m.c. 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.

[19] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. *SIGMOD Rec.*, 28(2):287–298, June 1999.

[20] Joseph M. Hellerstein, Ron Avnur, Andy Chou, Christian Hidber, Chris Olston, Vijayshankar Raman, Tali Roth, and Peter J. Haas. Interactive data analysis: The control project. *Computer*, 32(8):51–59, August 1999.

[21] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. *SIGMOD Rec.*, 26(2):171–182, June 1997.

[22] D.F. Jerding and J.T. Stasko. The information mural: a technique for displaying and navigating large information spaces. *Visualization and Computer Graphics, IEEE Transactions on*, 4(3):257–271, 1998.

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

[24] Zhicheng Liu, Biye Jiang, and Jeffrey Heer. immens: Real-time visual querying of big data. *Computer Graphics Forum (Proc. EuroVis)*, 32, 2013.

[25] William J. Schroeder, Ken Martin, and W.E. Lorensen. The visualization toolkit: An object-oriented approach to 3d graphics, fourth edition. 2004.

[26] Inc. SciDB. Scidb user's guide (version 13.3). 2013.

[27] Amy Henderson Squillacote. The paraview guide: A parallel visualization application. 2007.

[28] Chris Stolte and Pat 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.

[29] Michael Stonebraker, Jolly Chen, Nobuko Nathan, Caroline Paxson, and Jiang Wu. Tioga: Providing data management support for scientific visualization applications. In *Proceedings of the 19th International Conference on Very Large Data Bases*, VLDB '93, pages 25–38, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[30] H.T. Vo, J. Bronson, B. Summa, J.L.D. Comba, J. Freire, B. Howe, V. Pascucci, and C.T. Silva. Parallel visualization on large clusters using mapreduce. In *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, pages 81–88, 2011.