Visualizing Database Queries

by

Manasi Vartak

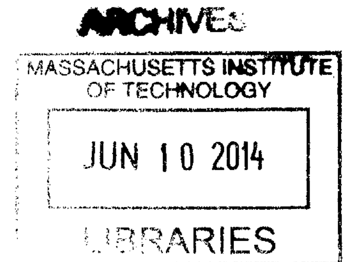
Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2014



© Massachusetts Institute of Technology 2014. All rights reserved.

Signature redacted

Author ....

Department of Electrical Engineering and Computer Science
May 20, 2014

Signature redacted

Certified by ....

Samuel Madden
Professor
Thesis Supervisor

Signature redacted

Accepted by ....

Leslie A. Kolodziejski
Chairman, Department Committee on Graduate Students

Visualizing Database Queries

by

Manasi Vartak

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2014, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

Data analysts operating on large volumes of data often rely on visualizations to interpret the results of queries. However, finding the right visualization for a query is a laborious and time-consuming task. We propose SEEDB, a system that partially automates this task: given a query, SEEDB explores the space of all possible visualizations, and automatically identifies and recommends to the analyst those visualizations it finds to be most "interesting" or "useful".

Thesis Supervisor: Samuel Madden
Title: Professor

Acknowledgments

This thesis wouldn't have been possible without the support of my research advisor, Sam Madden. His feedback and ideas have been crucial to the progress of this project. My thesis builds upon the original SeeDB work done by Aditya Parameswaran, and the work would not have been possible without his strong vision and drive. It has been a pleasure working with and learning from him.

I am grateful to Angela Zhang and Sashko Stubailo for their contributions to developing the SeeDB frontend. I would also like to thank other members of the MIT Database Group, particularly Mike Stonebraker, for his insights into this work and helpful guidance.

Most importantly, I'd like to thank my family for their constant encouragement and support throughout my graduate studies.

Contents

1 Introduction **9**

1.1 The data analysis process . . . 9

1.1.1 Example 1: Business Intelligence . . . 9

1.1.2 Example 2: Medical Data . . . 12

1.1.3 Example 3: Product Analysis . . . 14

1.1.4 SEEDB . . . 15

2 Problem Definition **17**

3 Architecture Overview **20**

4 SeeDB Frontend **22**

5 SeeDB Backend **24**

5.1 Basic Framework . . . 24

5.2 Optimizations . . . 25

5.2.1 Combine target and comparison view query . . . 25

5.2.2 Combine Multiple Aggregates . . . 26

5.2.3 Combine Multiple Group-bys . . . 26

5.2.4 Parallel Query Execution . . . 27

5.2.5 Sampling . . . 27

5.2.6 Pre-computing Comparison Views . . . 28

6 Experimental Evaluation **29**

6.1 Experimental Setup 29
6.2 Effect of Optimizations 30
6.2.1 Basic Framework 30
6.2.2 Combine target and comparison view query 31
6.2.3 Combine Multiple Aggregates 31
6.2.4 Parallel Query Execution 32
6.2.5 Combine Multiple Group-bys 33
6.3 Analytical Model 38
6.4 Choosing SEEDB parameters based on the model 41
6.5 Experimental Evaluation with All Optimizations 43

7 Related Work **45**
7.1 Interactive Data Visualization Tools 45
7.2 Data Cubes 46
7.3 General Purpose Data Analysis Tools 46
7.4 Multi-query optimization 46

8 Discussion **47**
8.1 Making SeeDB more efficient 47
8.2 Making SEEDB more flexible 48
8.3 Making SEEDB more helpful in data analysis 48

9 Conclusion **49**

List of Figures

1-1 Data: Total Sales by Store for Laserwave 11
1-2 Scenario A: Total Sales by Store 12
1-3 Scenario B: Total Sales by Store 13

3-1 SeeDB Architecture 20

4-1 SeeDB Frontend: Query Builder (left) and Example Visualizations (right) 23

6-1 Baseline Performance 31
6-2 Effect of Multiple Aggregate Optimization 32
6-3 Effect of Parallelism on Per View Execution Time 33
6-4 Effect of Parallelism on Total Execution Time 34
6-5 Average Temp Table Creation Time 35
6-6 Total Temp Table Creation Time 35
6-7 Average Temp Table Query Time 37
6-8 Total Temp Table Query Time 38
6-9 Total Execution Time 39
6-10 Total Execution Time vs. Number of Distinct Values 40
6-11 Actual and Estimated Temp Table Query Time 41
6-12 Actual and Estimated Temp Table Query Time 42
6-13 Actual and Estimated Total Execution Time 43
6-14 Performance Speedup With Optimizations 44

List of Tables

1.1 Data: Total Sales by Store for Laserwave 11

6.1 Datasets used for testing 29

6.2 Analytical model parameters 41

Chapter 1

Introduction

This thesis presents the design and implementation of a system, SeeDB, for automatically generating a large number of *interesting visualizations* for any given query. There are two key challenges in automatically generating visualizations: 1) automatically determining "interesting"-ness of a visualization, and 2) evaluating a large number of possible visualizations efficiently. In this work, we present solutions to both these problems.

1.1 The data analysis process

Data analysts must sift through very large volumes of data to identify trends, insights, or anomalies. Given the scale of data, and the relative ease and intuitiveness of examining data visually, analysts often use visualizations as a tool to identify these trends, insights, and anomalies. However, selecting the "right" visualization often remains a laborious and time-consuming task. We illustrate the data analysis process and challenges in choosing good visualizations using a few examples.

1.1.1 Example 1: Business Intelligence

Consider a dataset containing sales records for a nation-wide chain of stores. Let's say the store's data analyst is interested in examining how the newly-introduced heating

device, the "Laserwave Oven", has been doing over the past year. The results of this analysis will inform business decisions for the chain, including marketing strategies, and the introduction of a similar "Saberwave Oven".

The analysis workflow proceeds as follows: (1) The analyst poses a query to select the subset of data that she is interested in exploring. For instance, for the example above, she may issue the query:

$$\texttt{Q} = \texttt{SELECT * FROM Sales WHERE Product} = \text{“}\texttt{Laserwave}\text{''}$$

Notice that the results for this query may have (say) several million records each with several dozen attributes. Thus, directly perusing the query result is simply infeasible. (2) Next, the analyst studies various properties of the selected data by constructing diverse views or visualizations from the data. In this particular scenario, the analyst may want to study total sales by store, quantity in stock by region, or average profits by month. To construct these views, the analyst can use operations such as binning, grouping, and aggregation, and then generate visualizations from the view. For example, to generate the view 'total sales by store', the analyst would group each sales record based on the store where the sale took place and sum up the sale amounts per store. This operation can easily be expressed as the familiar aggregation over group-by query:

$$\texttt{Q}' = \texttt{SELECT store, SUM(amount) FROM Sales WHERE}$$
$$\texttt{Product} = \text{“}\texttt{Laserwave}\text{''}\ \texttt{GROUP BY store}$$

The result of the above query is a two-column table that can then be visualized as a bar-chart. Table 1.1 and Figure 1-1 respectively show an example of the results of this view and the associated visualization.

To explore the query results from different perspectives, the analyst generates a large number of views (and visualizations) of the form described above. (3) The analyst then manually examines each view and decides which ones are "interesting".

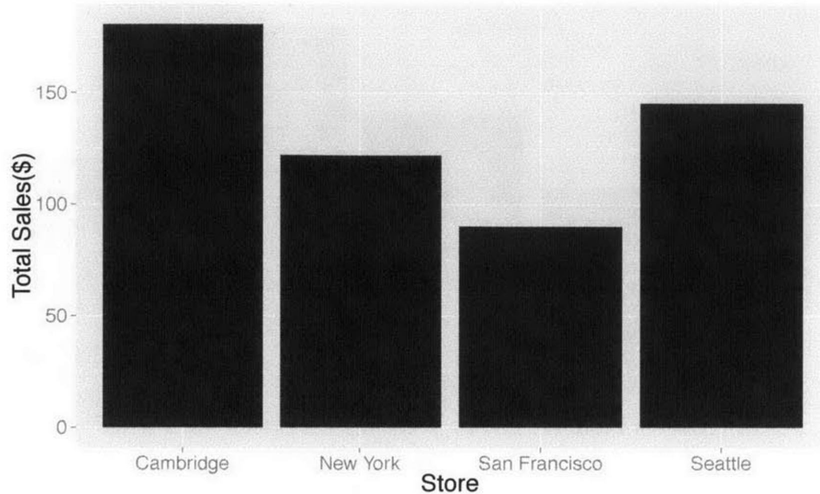


Figure 1-1: Data: Total Sales by Store for Laserwave

Store	Total Sales ($)
Cambridge, MA	180.55
Seattle, WA	145.50
New York, NY	122.00
San Francisco, CA	90.13

Table 1.1: Data: Total Sales by Store for Laserwave

This is a critical and time-consuming step. Naturally, what makes a view interesting depends on the application semantics and the trend we are comparing against. For instance, the view of Laserwave sales by store, as shown in Figure 1-1, may be interesting if the overall sales of all products show the *opposite* trend (e.g. Figure 1-2). However, the same view may be uninteresting if the sales of all products follow a similar trend (Figure 1-3). Thus, we posit that a view is *potentially "interesting" if it shows a trend in the subset of data selected by the analyst (i.e., Laserwave product-related data) that deviates from the equivalent trend in the overall dataset.* Of course, the analyst must decide if this deviation is truly an insight for this application. (4) Once the analyst has identified interesting views, the analyst may then either share these views with others, further interact with the displayed views (e.g., by drilling down or rolling up), or start afresh with a new query.

Of the four steps in the workflow described above, the ones that are especially

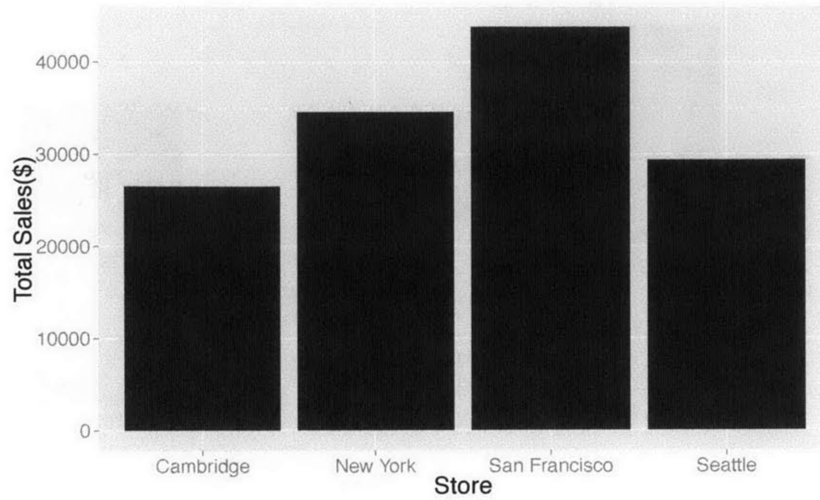


Figure 1-2: Scenario A: Total Sales by Store

repetitive and tedious are steps (2) and (3), where the analyst generates a large number of candidate views, and examines each of them in turn. The goal of our system, SEEDB, is to automate these labor-intensive steps of the workflow. Given a query Q indicating the subset of data that the analyst is interested in, SEEDB automatically *identifies and highlights to the analyst the most interesting views of the query results using methods based on deviation.* Specifically, SEEDB explores the space of all possible views and measures how much each view deviates from the corresponding view on the entire underlying dataset (e.g. Figure 1-1 vs. Figures 1-2 or 1-3.) By generating and scoring potential views automatically, SEEDB effectively eliminates steps (2) and (3) that the analyst currently performs. Instead, once SEEDB recommends interesting views, the analyst can evaluate this small subset of views using domain knowledge and limit further exploration to these views.

1.1.2 Example 2: Medical Data

Next, let us examine a use case in a completely different problem domain that also involves analyses similar to Example 1, and therefore, can benefit from the automatic construction of interesting views of a user query.

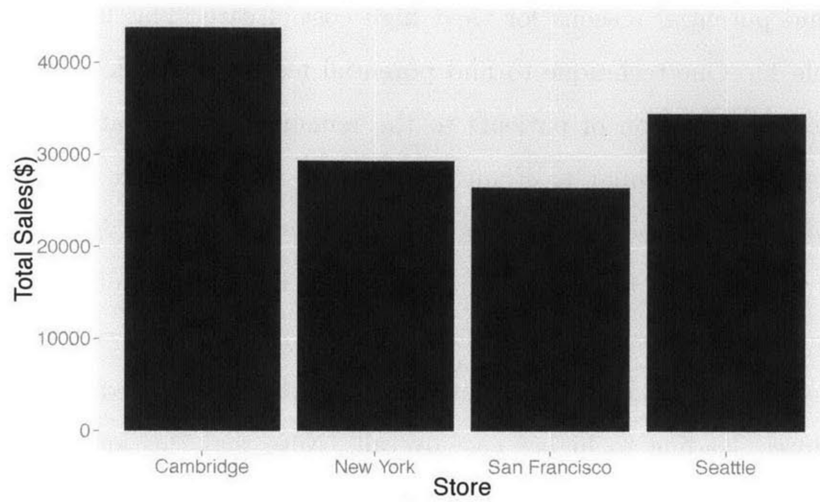


Figure 1-3: Scenario B: Total Sales by Store

Consider a medical researcher studying the cost of care for cancer patients[1]. The goal of the study is to identify patients whose cost of care is significantly greater than average and try to explain the high cost of care. Potential reasons for high cost include treatments for late-stage cancers, old age of the population, longer survival time (chemotherapy for a longer duration), type of treatment etc. Note that the goal is *not* to build a predictive model for cost; rather, it is to perform exploratory analysis to explain *why* certain patients have high cost of care. As a first pass, assume that the researcher identifies high cost patients as those with cost that is greater than two standard deviations away from the average cost. This can be expressed via the following SQL query, assuming a table of patients with their treatment costs and other treatment-related information.

```
Q2 = SELECT * FROM Patients where cost-
(SELECT AVG(cost) FROM Patients) >
2 * (SELECT STDDEV(cost) FROM Patients)
```

Once these patients have been identified, the researcher can begin to analyze the

[1]Real use scenario at an area hospital

data to find potential reasons for their high cost of care (This is similar to Step 1 in Example 1). One technique to find potential reasons for high cost is to compare the high-cost population of patients to the remaining set of patients (called "low-cost" population). Similar to Example 1 above, we posit that the *characteristics that explain high cost are precisely those characteristics that are different between the high-cost and low-cost population.* For instance, if the majority of high-cost patients were those with late-stage disease (sicker patients) while the low-cost patients were not, the researcher could reason that the sicker patients needed more medications or procedures, leading to higher cost overall. Note that this analysis is similar to Example 1 where we compared the "total sales by year" for *Laserwave oven* vs. all store products. In this setup, we want to compare the "total patients by disease stage" for high-cost patients vs. low-cost patients. The only change in the problem formulation is that we are now comparing views of query Q2 against views of the remaining table, instead of views over the entire table. The rest of the framework remains unchanged. Therefore, SEEDB can be used to automatically construct a large number of views of the high-cost population, compare each view to the corresponding view over the low-cost population, and identify views showing the highest difference between the two populations. These views identify potential causes for high cost. Once the researcher has identified views of interest, he or she can follow up with more complex analyses like statistical significance testing or machine learning.

1.1.3 Example 3: Product Analysis

Consider a company like Facebook[2] that continuously deploys changes to its website and mobile apps, and tracks user interaction through detailed logging. Product specialists at Facebook use these logs to study how different Facebook users respond to changes to the web or mobile experience [2]. Each user can be characterized by a large number of features such as location, age, device used, number of friends etc. For each user, the logs note the actions taken on the website or app such as likes,

[2]www.facebook.com

shares, comments, page visits etc. For example, in order to study the user response to an update to the mobile app, the specialist compares user interaction metrics for a large number of mobile users before and after the update (usually a week on week comparison). The goal is to find patterns in the interaction metrics based on different user characteristics. Therefore, if it appears that the average number of app visits on the iOS app have reduced signficantly following the update, it would indicate a problem with the iOS app.

In terms of the SEEDB framework, log data from the week following the app update constitutes the query results we seek to analyze and the log data from the prior week comprises the comparison dataset. As in Examples 1 and 2, the analysis process requires the specialist to create diverse views of the query results, compare views to equivalent views on the comparison dataset, and pick the views showing the highest difference. SEEDB can therefore be used to automate this process and surface only the most interesting views.

1.1.4 SEEDB

This thesis describes a prototype of the SEEDB system that finds interesting visualizations of query results. Given a query Q indicating the subset of data that the analyst is interested in, SEEDB automatically *identifies and highlights to the analyst the most interesting views of the query results using methods based on deviation.* Our prototype is based on [12] which describes the vision for SEEDB.

To efficiently and accurately identify interesting views of any given query, we are faced with the following challenges:

- We must determine metrics that accurately measure the "deviation" of a view with respect to the equivalent view on the entire database (e.g., Figure 1-1 vs. 1-2), while simultaneouly ensuring that SEEDB is not tied to any particular metric(s)

- We must intelligently explore the space of candidate views. Since the number of candidate views (or visualizations) increases as the square of the number of

attributes in a table (we will demonstrate this in subsequent sections), generating and evaluating all views, even for a moderately sized dataset (e.g. 1M rows, 100 attributes), can be prohibitively expensive

- While executing queries corresponding to different views, we must share computation as much as possible. For example, we can compute multiple views and measure their deviation all together in one query. Independent execution, on the other hand, will be expensive and wasteful
- Since analysis must happen in real-time, we must trade-off accuracy of visualizations or estimation of "interestingness" for reduced latency.

Contributions of the work described in this thesis are:

- We implement the SEEDB system based on [12] to automatically find interesting views of queries (Chapter 3).
- We propose and implement a number of optimizations to efficiently perform the search for interesting views and share computation between multiple views simultaneously (Chapter 5.2).
- We evaluate the performance of our optimizations on a range of datasets and demonstrate the resulting 100x speedup (Chapter 6.2).
- We model the performance of SEEDB in terms of various parameters of SEEDB and the underlying database, and use this model to identify optimal parameter settings for SEEDB (Chapter 6.3).

Chapter 2

Problem Definition

Now that we have defined the goal of SEEDB, we formally describe the problem that SEEDB seeks to address. Given a database D and a query Q, SEEDB considers a number of views that can be generated from Q by adding relational operators to group and aggregate query results. For the purpose of this discussion, we will refer to views and visualizations interchangeably, since it is straightforward to translate views into visualizations automatically. For example, there are straightforward rules that dictate how the view in Table 1.1 (Example 1) can be transformed to give a visualization like Figure 1-1. Furthermore, we limit the set of candidate views to those that generate a two-column result via a single-attribute grouping and aggregation (e.g. Table 1.1). However, SEEDB techniques can directly be used to recommend visualizations for multiple column views (> 2 columns) that are generated via multi-attribute grouping and aggregation.

We consider a database D with a snowflake schema, with dimension attributes A, measure attributes M, and potential aggregate functions F over the measure attributes. We limit the class of queries Q posed over D to be those that select one or more rows from the fact table, and denote the results as D_Q.

Given such a query Q, SEEDB considers all views V_i that perform a single-attribute group-by and aggregation on D_Q. We represent V_i as a triple (a, m, f), where $m \in M, a \in A, f \in F$, i.e., the view performs a group-by on a and applies the

aggregation function f on a measure attribute m. We call this the *target view*.

$$\texttt{SELECT } a, f(m) \texttt{ FROM } D_Q \texttt{ GROUP BY } a$$

As discussed in the previous section, SEEDB evaluates whether a view V_i is interesting by computing the deviation between the view applied to the selected data (i.e., D_Q) and the view applied to the entire database. The equivalent view on the entire database $V_i(D)$ can be expressed as shown below that we call the *comparison view*.

$$\texttt{SELECT } a, f(m) \texttt{ FROM } D \texttt{ GROUP BY } a$$

The results of both the above views are tables with two columns, namely a and $f(m)$. We normalize each result table into a probability distribution, such that the values of $f(m)$ sum to 1. For our example in Table 1.1, the probability distribution of $V_i(D_Q)$, denoted as $P[V_i(D_Q)]$, is: (Jan: 180.55/538.18, Feb: 145.50/538.18, March: 122.00/538.18, April: 90.13/538.18). A similar probability distribution can be derived for $P[V_i(D)]$.

Given a view V_i and probability distributions for the target view ($P[V_i(D_Q)]$) and comparison view ($P[V_i(D)]$), the *utility* of V_i is defined as the distance between these two probability distributions. Formally, if S is a distance function,

$$U(V_i) = S(P[V_i(D_Q)], P[V_i(D)])$$

The utility of a view is our measure for whether the target view is "potentially interesting" as compared to the comparison view: the higher the utility, the more the deviation from the comparison view, and the more likely the view is to be interesting. Computing distance between probability distributions has been well studied, and SEEDB supports a variety of metrics to compute utility. These include:

- **Earth Movers Distance (EMD)** [22]: Commonly used to measure differences between color histograms from images, EMD is a popular metric for comparing discrete distributions.

- **Euclidean Distance**: The L2 norm or Euclidean distance considers the two distributions to be points in a high dimensional space and measures the distance between them.
- **Kullback-Leibler Divergence**(K-L divergence) [21]: K-L divergence measures the information lost when one probability distribution is used to approximate the other one.
- **Jenson-Shannon Distance** [20, 19]: Based on the K-L divergence, this distance measures the similarity between two probability distributions.

While we set the default SeeDB distance metric as EMD (due to its simplicity), users can choose to use any of the distance metrics defined above. We note that the above definition of a view and its utility is merely one of many possible definitions and we choose this particular definition for simplicity and its intuitive nature. We describe other view definitions and utility metrics in Section 8.

Problem 2.1 *Given an analyst-specified query Q on a database D, a distance function S, and a positive integer k, find k views $V \equiv (a, m, f)$ that have the largest values of $U(V)$ among all the views that can be represented using a triple (a, m, f), while minimizing total computation time.*

While the problem definition above assumes that we have been provided with a query Q and we compare views on Q with corresponding views on the entire database D, other settings are also possible. For instance, in Example 2 (medical research), we are provided with query Q and we must compare views on it to the remaining dataset, i.e. $D \backslash Q$. Similarly, in Example 3 (product analysis), we are comparing equivalent views on two separate queries $Q1$ and $Q2$. The SeeDB framework is agnostic to where the comparison dataset is coming from and its contents. It merely provides a point of comparison for the target view, and therefore SeeDB can be used unchanged for any of these settings.

Chapter 3

Architecture Overview

In this chapter, we present an overview of the SEEDB architecture and relevant design decisions. Our SEEDB implementation is designed as a layer on top of a traditional relational database system. While optimization opportunities are restricted by virtue of being outside the database, our design permits SEEDB to be used in conjunction with a variety of existing database systems. SEEDB is comprised of two parts: a frontend and a backend. The frontend is a "thin client" that is used to issue queries, display visualizations and allow basic interactions with the visualizations. The backend, in contrast, performs all the computation required to generate and select views to be recommended. Figure 3-1 depicts the architecture of our system.

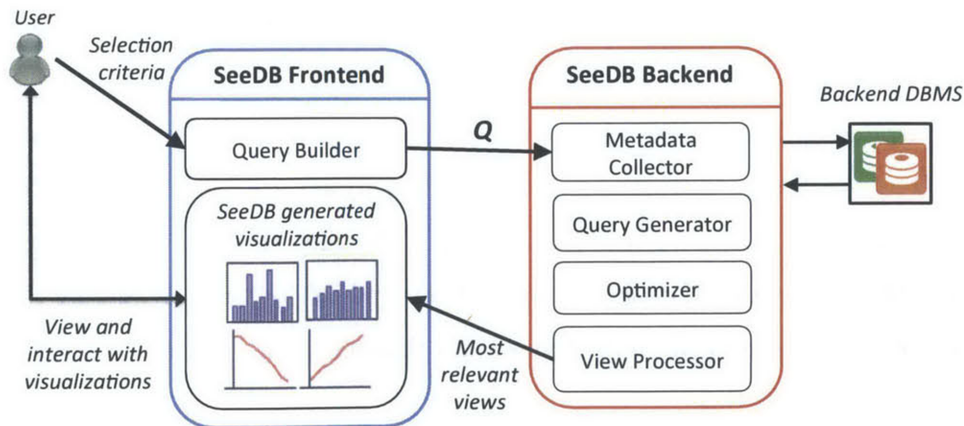


Figure 3-1: SeeDB Architecture

An analyst uses the frontend to issue queries to SEEDB. We provide three mech-

anisms for the analyst to issue queries (further discussion in Chapter 4). Once the analyst issues a query via the frontend, the backend takes over. First, the Metadata Collector module queries metadata tables (a combination of database-provided and SEEDB specific tables) for information such as table sizes, column types, data distribution, and table access patterns. The resulting metadata along with the analyst's query is then passed to the Query Generator module. The purpose of the Query Generator is two-fold: first, it uses metadata to prune the space of candidate views to only retain the most promising ones; and second, it generates target and comparison views for each view that has not been pruned. The SQL queries corresponding to the target and comparison views are then passed to the Optimizer module. We refer to these queries collectively as *view queries*. Next, the Optimizer module determines the best way to combine view queries intelligently so that the total execution time is minimized. (We discuss optimizations performed by SEEDB in Chapter 5.) Once the Optimizer module has generated the optimized queries, SEEDB runs them on the underlying DBMS. Results of the optimized queries are processed by the View Processor in a streaming fashion to produce results for individual views. Individual view results are then normalized and the utility of each view is computed. Finally SEEDB selects the top k views with the highest utility and returns them to the SEEDB frontend. The frontend generates and displays visualizations for each of these view.

Chapter 4

SeeDB Frontend

The SEEDB frontend, designed as a thin client, performs two main functions: it allows the analyst to issue a query to SEEDB, and it visualizes the results (views) produced by the SEEDB backend. To provide the analyst maximum flexibility in issuing queries, SEEDB provides the analyst with three mechanisms for specifying an input query: (a) directly filling in SQL into a text box, (b) using a query builder tool that allows analysts unfamiliar with SQL to formulate queries through a form-based interface, and (c) using pre-defined query templates which encode commonly performed operations, e.g., selecting outliers in a particular column.

Once the analyst issues a query via the SEEDB frontend, the backend evaluates various views and delivers the most interesting ones (based on utility) to the frontend. For each view delivered by the backend, the frontend creates a visualization based on parameters such as the data type (e.g. ordinal, numeric), number of distinct values, and semantics (e.g. geography vs. time series). The resulting set of visualizations is displayed to the analyst who can then easily examine these "most interesting" views at a glance, explore specific views in detail via drill-downs, and study metadata for each view (e.g. size of result, sample data, value with maximum change and other statistics). Figure 4-1 shows a screenshot of the SEEDB frontend (showing the query builder) in action.

SQL Visual Selection File Upload

Database: donations

Table: donations_small

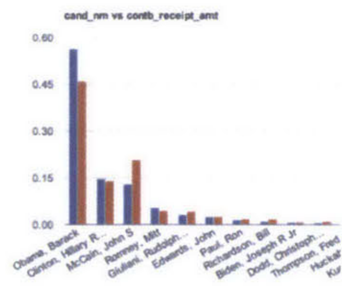
Predicates

	Column Name	Operator	Value
where	cand_nm	=	Obama, Barack
where	cand_nm	in	'SAN FRANCISCO',

+ Add Predicate

Distributions in data set

In the case of many unique values, only the 20 most common are displayed.



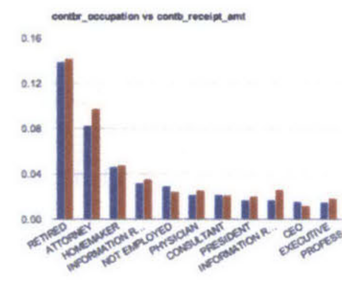


Figure 4-1: SeeDB Frontend: Query Builder (left) and Example Visualizations (right)

Chapter 5

SeeDB Backend

The SEEDB backend is responsible for all the computations for generating and selecting views. To achieve its goal of finding the most interesting views accurately and efficiently, the SEEDB backend must not only accurately estimate the quality of a large number of views but also minimize total processing time. We first describe the basic SEEDB backend framework and then discuss our optimizations.

5.1 Basic Framework

Given a user query Q, the basic approach computes all possible two-column views obtained by adding a single-attribute aggregate and group-by clause to Q. (Remember from 2 that Q is any query that selects one or more rows from the underlying table.) The target and comparison views corresponding to each view are then computed and each view query is executed independently on the DBMS. The query results for each view are normalized, and utility is computed as the distance between these two distributions (Section 2). Finally, the top-k views with the largest utility are chosen to be displayed. If the underlying table has d dimension attributes and m measure attributes, $2 * d * m$ queries must be separately executed and their results processed. Even for modest size tables (1M tuples, d=50, m=5), this technique takes prohibitively long (700s on Postgres). The basic approach is clearly inefficient since it examines every possible view and executes each view query independently. We next

discuss how our optimizations fix these problems.

5.2 Optimizations

Since view queries tend to be very similar in structure (they differ in the aggregation attribute, grouping attribute or subset of data queried), SeeDB uses multiple techniques to intelligently combine view queries. In addition, SeeDB leverages parallelism and partitioning to further reduce query execution time. The ultimate goal of these optimizations is to minimize scans of the underlying dataset by sharing as many table scans as possible. SeeDB supports the following optimizations as well as their combinations.

5.2.1 Combine target and comparison view query

Since the target view and comparison views only differ in the subset of data that the query is executed on, we can easily rewrite these two view queries as one. For instance, for the target and comparison view queries $Q1$ and $Q2$ shown below, we can add a group by clause to combine the two queries into $Q3$.

$$
\begin{aligned}
Q1 =& \texttt{SELECT } a, f(m) \texttt{ FROM } D \texttt{ WHERE x < 10 GROUP BY } a \\
Q2 =& \texttt{SELECT } a, f(m) \texttt{ FROM } D \texttt{ GROUP BY } a \\
Q3 =& \texttt{SELECT } a, f(m), \texttt{CASE IF x < 10 THEN 1 ELSE 0 END } as\ group1,\ 1\ as\ group2 \\
& \texttt{FROM } D \texttt{ GROUP BY } a,\ group1,\ group2
\end{aligned}
$$

This rewriting allows us to obtain results for both queries in a single table scan. The impact of this optimization will depend on the selectivity of the input query and the presence of indexes. When the input query is less selective, the query executor must do more work in running the two queries separately. In contrast, if the target and comparison views are both selective, and an index is present on their selection attributes, individual queries can run much faster than the combined query which must scan the entire table.

5.2.2 Combine Multiple Aggregates

A large number of view queries have the same group-by attribute but different aggregation attributes. In addition, the majority of real-world datasets, tables have few measure attributes but a large number of dimension attributes (e.g. the Super Store dataset has 5 measure attributes but tens of dimension attributes). Therefore, SEEDB combines all view queries with the same group-by attribute into a single, combined view query. For instance, instead of executing queries for views (a_1, m_1, f_1), (a_1, m_2, f_2) ... (a_1, m_k, f_k) independently, we can combine the n views into a single view represented by ($a_1, \{m_1, m_2 \ldots m_k\}, \{f_1, f_2 \ldots f_k\}$). We expect this optimization to offer a speed-up roughly linear in the number of measure attributes.

5.2.3 Combine Multiple Group-bys

Since SEEDB computes a large number of group-bys, one significant optimization is to combine queries with different group-by attributes into a single query with multiple group-bys attributes. For instance, instead of executing queries for views (a_1, m_1, f_1), (a_2, m_1, f_1) ... (a_n, m_1, f_1) independently, we can combine the n views into a single view represented by ($\{a_1, a_2 \ldots a_n\}$, m_1, f_1) and post-process results at the backend. Alternatively, if the SQL GROUPING SETS[1] functionality is available in the underlying DBMS, SEEDB can leverage that as well. While this optimization has the potential to significantly reduce query execution time, the number of views that can be combined will depend on the number of distinct groups present for the given combination of grouping attributes. For a large number of distinct groups, the query executor must keep track of a large number of aggregates. This increases computational time as well as temporary storage requirements, making this technique ineffective. The number of distinct groups in turn depends on the correlation between values of attributes that are being grouped together. For instance, if two dimension attributes a_i and a_j have n_i and n_j distinct values respectively and a correlation

[1]GROUPING SETS allow the simultaneous grouping of query results by multiple sets of attributes.

coefficient of c, the number of distinct groups when grouping by both a_i and a_j can be approximated by n_i*n_j*(1-c) for $c \neq 1$ and n_i for c=1 (n_i must be equal to n_j in this case). As a result, we must combine group-by attributes such that the number of distinct groups remains small enough. In Section 6.2, we characterize the performance of this optimization and devise strategies to choose dimension attributes that can be grouped together.

If we choose a set of grouping attributes that creates a large number of distinct groups, not only does the query executor need to do more work, the result returned to the client is large and the client takes longer to process the result. Since this process can be very inefficient, we choose to store the intermediate results as temporary tables and then subsequently query the temp tables to obtain the final results. For ease of further analysis, we denote these two phases as *Temp Table Creation* (where the intermediate results are created and stored) and *Temp Table Querying* (where the temp tables are queried for final results) respectively.

5.2.4 Parallel Query Execution

While the above optimizations reduce the number of queries executed, we can further speedup SEEDB processing by executing view queries in parallel. When executing queries in parallel, we expect co-executing queries to share pages in the buffer pool for scans of the same table, thus reducing the total execution time. However, a large number of parallel queries can lead to poor performance for several reasons including buffer pool contention, locking and cache line contention [13]. As a result, we must identify the optimal number of parallel queries for our workload.

5.2.5 Sampling

For large datasets, sampling can be used to significantly improve performance. To use sampling with SEEDB, we precompute a sample of the entire dataset (size of sample depends on desired accuracy). When a query is issued to SEEDB, we run all view queries against the sample and pick the top-k views. Only these high-utility views

are then computed on the entire dataset. As expected, the accuracy of views depends on the size of the sample; a larger sample generally produces more accurate results and we can develop bounds on the accuracy of aggregates computed on samples. There are two ways to employ sampling in the SeeDB setting: (1) depending on the response time required, choose a sample size that will provide the required response time and accordingly return to the user the estimated accuracy of the results; or (2) given a user-specific threshold for accuracy, determine the correct size of the sample and apply the above technique.

5.2.6 Pre-computing Comparison Views

We notice that in the case where our comparison view is constructed from the entire underlying table (Example 1 in Chapter 1), comparison views are the same irrespective of the input query. In this case, we can precompute all possible comparison views once and store them for use in all future comparisons. If the dataset has d dimension and m measure attributes, pre-computing comparison views would add $d*m$ tables. This corresponds to an extra storage of $O(d*m*n)$ where n is the maximum number of distinct values in any of the d attributes. In this case, we still need to evaluate each target view, and we can leverage previous optimizations to speed up target view generation.

Note that pre-computation cannot be used in situations where the comparison view depends on the target view (Example 2) or is directly specified by the user (Example 3).

Chapter 6

Experimental Evaluation

We evaluated the performance of SEEDB on a variety of datasets with different sizes and number of attributes. We now discuss in detail the effect of each of our optimization strategies and develop an analytical model that lets us pick the optimial settings of SEEDB parameters.

6.1 Experimental Setup

Type	Dataset Name	Num Rows	Num Dimensions	Num Measures	Size (GB)	Num Views
Small	$Small_1$	1M	5	2	0.1	10
	$Small_2$	10M	5	2	1	
	$Small_3$	100M	5	2	10	
Medium	Med_1	1M	50	5	0.4	250
	Med_2	10M	50	5	4	
	Med_3	100M	50	5	40	
Large	$Large_1$	1M	100	10	1	1000
	$Large_2$	10M	100	10	10	
	$Large_3$	100M	100	10	100	

Table 6.1: Datasets used for testing

Table 6.1 lists the datasets on which we evaluated the performance of SEEDB. These datasets are synthetically generated and their size varies from 100 MB to 100 GB and number of attributes ranges from 5 - 100 dimension attributes and 2 - 10

measure attributes. The relative cardinality of dimension and measure attributes was chosen to model real-world datasets which usually have a large number of dimension attributes but few measure attributes. To accurately estimate the effect of specific optimizations, for each dataset, we also created a supplemental dataset with the same specifications except that each attribute had the same number of distinct values (100).

For each dataset, we used SEEDB to find the top 20 views of the input query. Any run that took more than 1 hour was terminated. All experiments were repeated three times and the results were averaged. We ran experiments using Postgres as the backend database for SEEDB and a single machine with 32 Intel Xeon E7 processors with hyperthreading enabled and 256 GB RAM.

6.2 Effect of Optimizations

We now present experimental characterization of the optimization strategies described in 5. Our goal is to understand the effect of each strategy on SEEDB performance in order to build an analytical model and predict the optimal set of parameters for SEEDB.

6.2.1 Basic Framework

We first examined the baseline performance of SEEDB without any optimizations. For each possible view, we executed the target and comparison view queries separately and sequentially, and then picked the top views. This corresponds to the basic framework described in Section 5.1. The number of queries executed for each dataset was twice the number of views shown in Table 6.1. Figure 6-1 shows the baseline performance for Small, Medium and Large datasets of size 1M. We observe that execution time increases super-linearly as the size of the dataset (number of dimension and measure attributes) increases. Moreover, as mentioned before, even for the Medium sized dataset (1M rows, 5 measure and 50 dimension attributes), SEEDB execution takes 700s, a latency that is unacceptable for interactive queries.

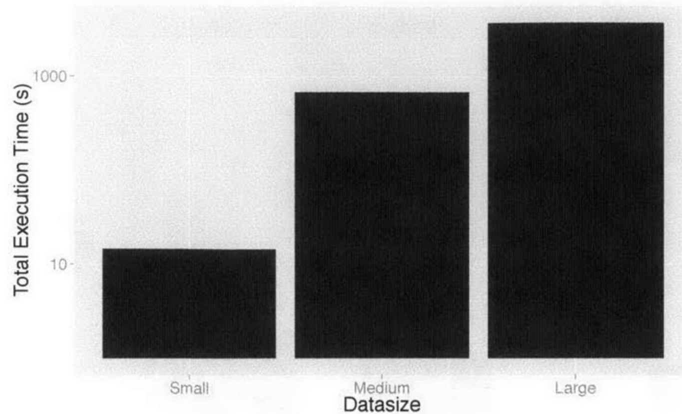


Figure 6-1: Baseline Performance

6.2.2 Combine target and comparison view query

Next, we study the effect of combining target and comparison view queries as described in Section 5.2.1. The goal of this optimization is to execute both queries in a single scan of the table. Therefore, the total number of queries executed is equal to the number of views possible for a given dataset. This optimization offers an average speed up of 1.7x across a range of selectivities for the input query.

6.2.3 Combine Multiple Aggregates

SeeDB uses the parameter n_{agg} to denote the number of aggregates that may be included in the same view query. Therefore, given a set of view queries with the same group-by attribute, view queries are combined so that each query has up to n_{agg} aggregates. We varied $n_{agg} \in 2, 3, 5, 10$ for each dataset (Note that the Small and Medium dataset have only 2 and 5 measure attributes respectively). Figure 6-2 shows the performance gains achieved for the 1M row datasets. We see that for a given dataset, increasing n_{agg}, i.e. computing more aggregates in the same query, gives an almost linear speedup. We also notice that this optimization is slightly more effective for larger datasets.

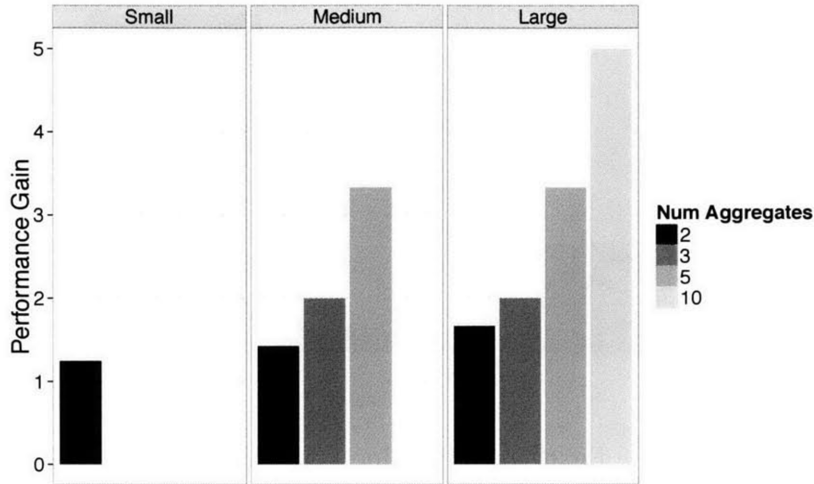


Figure 6-2: Effect of Multiple Aggregate Optimization

6.2.4 Parallel Query Execution

As discussed in Section 5.2.4, executing view queries in parallel can provide significant performance gains; however, a high degree of parallelism can lead to a performance drop off for several reasons. Potential reasons include disk contention, RAM usage, lock contention, context switches and cache line contention [13]. Identifying the right amount of parallelism requires tuning for the particular workload. The SEEDB workload consists of multiple parallel queries performing full sequential scans of a given table. To evaluate the effect of parallelism, we varied the number of queries that can be executed in parallel and measured its effect on the average time to execute a query as well as the total execution time. Since our backend DBMS is Postgres, parallel query execution is implemented by opening multiple connections and running queries sequentially on each connection.

Figure 6-3 shows the effect of parallelism on the average execution time per view (Medium dataset, 1M rows). Note the log scale on the y-axis. We observe that query execution time stays flat between 1 - 10 connections, suddenly increases between 10 - 20 connections, and then increases linearly for more than 20 connections. This suggests that the benefits of parallel execution are outweighed by contention beyond 20 connections. Figure 6-4 shows the total time (as opposed to per view execution

time) taken by SeeDB for varying levels of parallelism. We observe that the minima occurs in the range between 10 - 20 parallel queries and the execution times flatten out after 40 parallel queries. This trend is the effect of two opposing factors: (A) increased parallelism increases contention, and therefore increases per query execution time, and (B) parallelism decreases the number of batches of queries that must be executed, thus reducing overall time. We will take these two opposing forces into account when we develop an analytical model for SeeDB execution time in Section 6.3.

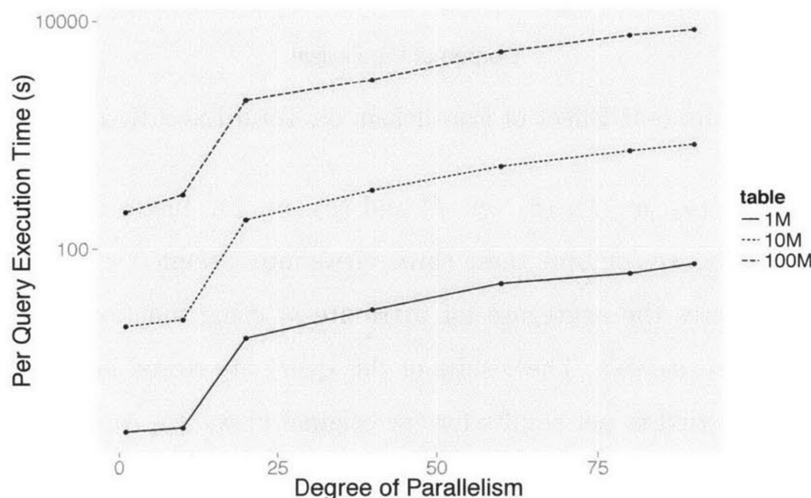


Figure 6-3: Effect of Parallelism on Per View Execution Time

6.2.5 Combine Multiple Group-bys

Combining multiple attributes in a group-by clause can reduce the number of views that must be explored individually. However, the benefits of this optimization can be lost to high intermediate result cost if the number of distinct groups is large. As mentioned before, we divide this optimization into two phases: (1) *Temp Table Creation* and (2) *Temp Table Querying*. In the first phase, we run queries with grouping based on multiple attributes and store the results in temporary tables. In the second phase, we run single aggregate+group-by queries on the temp tables to obtain final results for views. For instance, suppose that we want to compute the

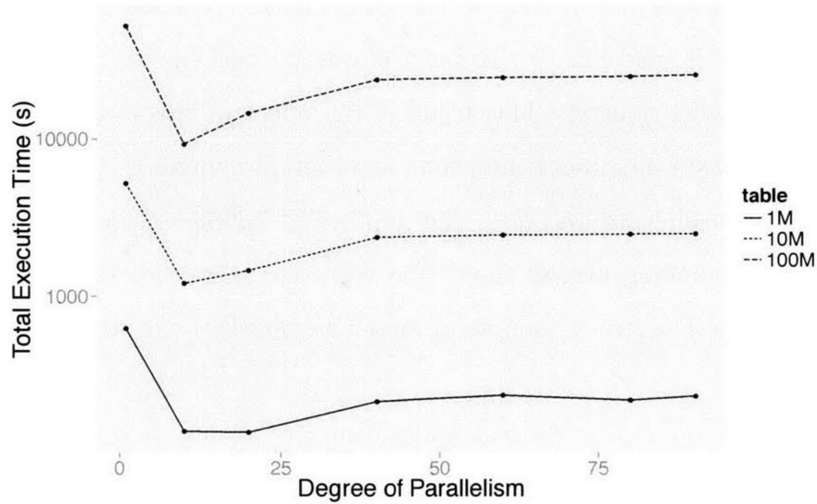


Figure 6-4: Effect of Parallelism on Total Execution Time

results for views (a_1, m, f), (a_2, m, f) and (a_3, m, f). Instead of executing these views individually, we combine these three views into a single view, ($\{a_1, a_2, a_3\}$, m, f), which computes the aggregate for attribute m using function f and groups by the attributes $\{a_1, a_2, a_3\}$. The results of this query are stored in a temporary table which is then queried to get results for the original views, (a_1, m, f), (a_2, m, f) and (a_3, m, f).

SeeDB uses the parameter n_{GB} to denote the number of attributes in the group-by clause. To evaluate the effect of combining group-bys, we ran SeeDB by varying number of group-by attributes, i.e. the n_{GB} parameter, between 1 and the number of dimensions d in a given table. For each run, we measured the amount of time taken to create the temporary tables, the time taken to query the temporary tables and the total execution time. Figure 6-5 to 6-9 show the results for the Medium dataset of size 1M tuples.

Temp Table Creation Phase: Figure 6-5 shows the average time required to create temporary tables for n_{GB}=1...50. There are several points to note in this graph: (1) for $n_{GB} >= 10$, the number of connections does not have a significant impact of the temp table creation time. We see this behavior because for $n_{GB} >= 10$, the number of temp tables created is $<= 5$ and therefore a maximum of 5 connections

is used irrespective of the number of connections that are open. We also note an upward trend in the total temp table creation time with increasing n_{GB} because the temporary tables gradually become larger in size (the number of rows in the table is bounded by the number of rows in the input table but an increase in n_{GB} increases the columns present in the table). We also observe that the "sweet spot" for temp table creation occurs between 1 - 2 group-bys.

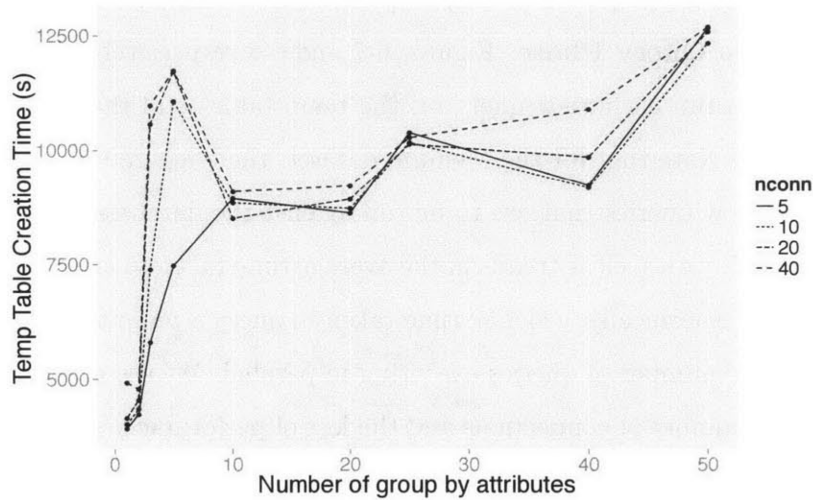


Figure 6-5: Average Temp Table Creation Time

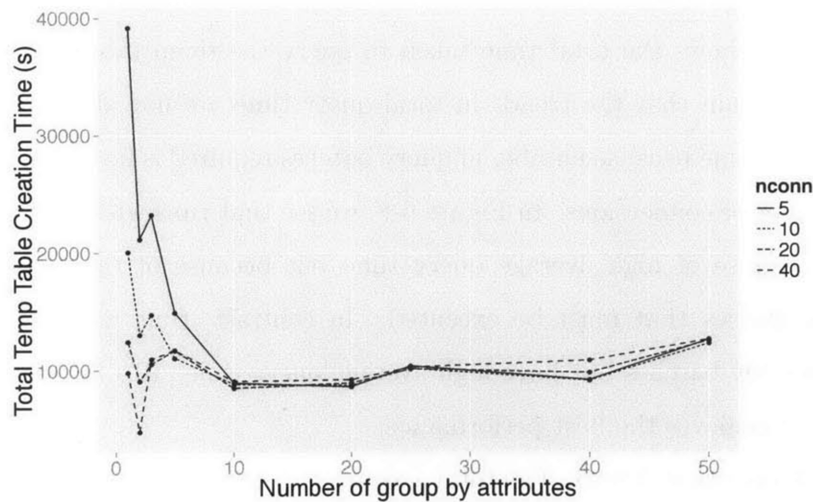


Figure 6-6: Total Temp Table Creation Time

Figure 6-6 shows the total time spent in creating temporary tables in SEEDB. As before, we see that the total time flattens out after 10 group-bys; however, we observe a reordering of the the trend lines with respect to number of connections. While the average time taken to generate temp tables with 5 connections is the least, SEEDB must run more batches of queries to create the required number of temporary tables. We observe that that 40 connections is optimal for minimizing the total temp table creation time. As in the previous diagram, we observe a minima around $n_{GB} = 2$.

Temp Table Query Phase: Figures 6-7 and 6-8 respectively show the average time required to run each view query on the temp tables and the total time to run all view queries. Note that for the Medium dataset, there are 250 possible views and therefore 500 view queries that are to be run against the database.

In Figure 6-7, we see clear trends in the average time taken to execute view queries on temp tables. Specifically: (1) The time taken to query a temp table increases non-linearly with the number of queries executing in parallel. We see that the trend lines are ordered by number of connections and the loss of performance grows with number of connections. (2) As before, we observe a slight increase in the execution time as the size of temp tables increases. This is not surprising since the query executor must scan and process more data. (3) Finally, we observe a minima at $n_{GB} = 2$, similar the to two graphs above.

Figure 6-8 shows the total time taken to query the temp tables for all the final views. Note again that the trends in total query time are not identical to those in average query time because number of query batches required is inversely proportional to the number of connections. In Figure 6-8, we see that runs with 5 connections are slow not because of high average query time but because of the large number of batches of queries that must be executed. In contrast, runs with 40 connections require very few batches but have high average query time. 10 - 20 connections and $n_{GB} = 1 - 2$ achieves the best performance.

Total Execution Time: The total execution time for SEEDB is the sum of time required for the two phases above. Figure 6-9 shows the total SEEDB execution time for different values of n_{GB} and number of connections. We observe that the best

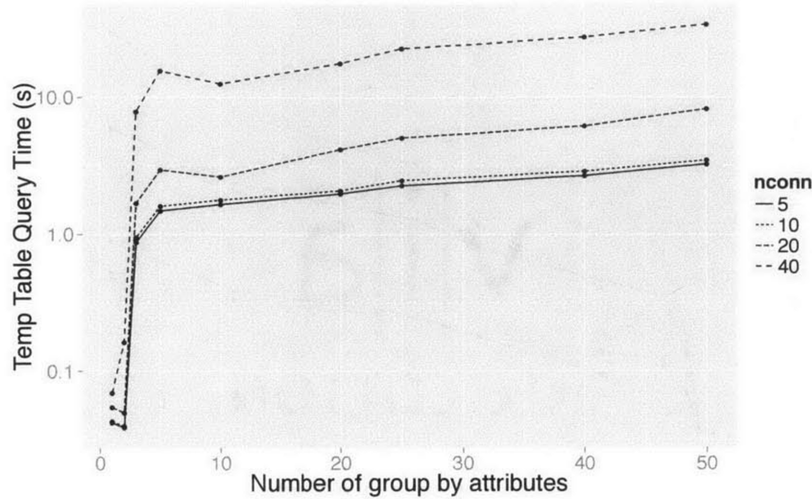


Figure 6-7: Average Temp Table Query Time

performance is obtained for $n_{GB} = 2$ and 40 parallel connections.

Effect of Number of Groups: The above experiments suggest that $n_{GB} = 2$ is the optimal value for the number of group by attributes, both for temp table creation and querying. Next we study whether this constraint applies to the number of attributes in the group by clause or the number of distinct groups produced by the grouping. For this purpose, we created variants of the Medium dataset (1M rows) where each dimension attribute had n distinct values with n=10...1000. We then repeated the experiments combining multiple group-bys using these datasets. Figure 6-10 shows the results of this test. In the test dataset, the total number of distinct groups for attributes a_i and a_j is the product of the number of distinct groups for each attribute. We observe in 6-10 that the previously-observed minima at $n_{GB} = 2$ is actually a function of the number of distinct groups that are generated by the multiple-attribute grouping. Specifically, we observe that the optimal value for the number of distinct groups is in the range of 10,000 - 100,000. We observe similar trends for temp table creation and query time.

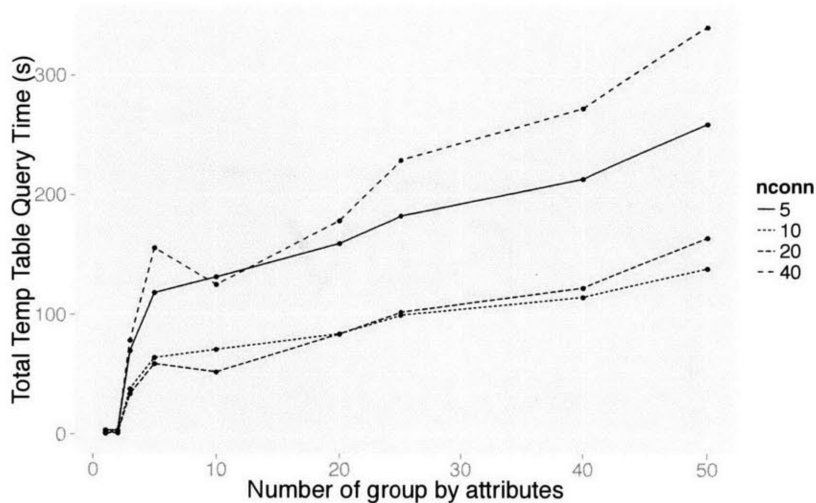


Figure 6-8: Total Temp Table Query Time

6.3 Analytical Model

In this section, we use insights from the experimental characterization of various optimizations to develop an analytical model of SeeDB performance. Table 6.2 defines the various parameters used in our model.

As before, we break our model into two parts, *Temp Table Creation* or *Phase 1* and *Temp Table Querying* or *Phase 2*. Note further that in each of these phases, multiple queries are executing in parallel. We call the set of queries executing in parallel as a "batch" of queries. Suppose that the total number of queries to be executed is q and n_{conn} queries can be executed in parallel. Then the total number of batches required to executed all the queries is $\frac{q}{n_{conn}}$. We denote the number of query batches used in temp table creation and querying as b_{create} and b_{query} respectively.

We now describe the analytical model for Phase 1 or *Temp Table Creation*. The time required to create a temp table is proportional to the sum of the time required to query the input table, aggregate measure attributes and finally write the temp table. We claim that the time taken to process one row of any table is equal to the constant time to process any record plus the time required to process all the columns

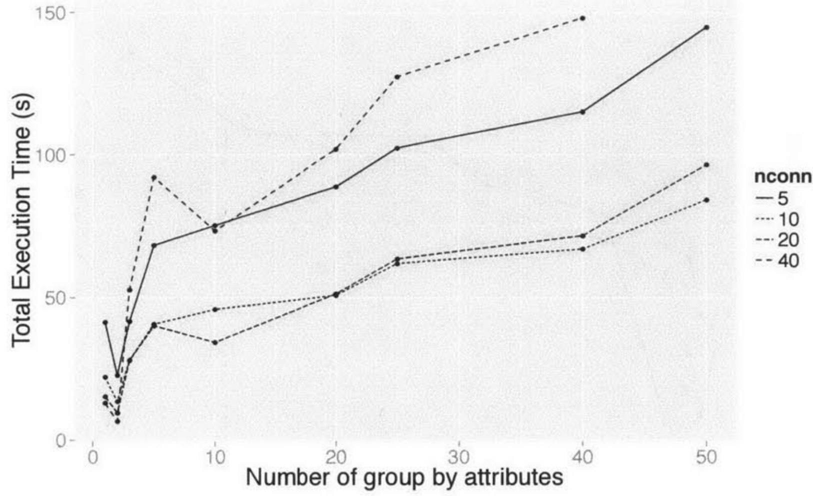


Figure 6-9: Total Execution Time

in the record.

$$T_{create} \quad = \quad n * \frac{(t_r \; + \; t_c * c)}{n_{conn}} \; + \; t_w * n_{tt} * (t_r \; + \; t_c * c_{tt})$$

We can model *Temp Table Querying* similarly. Total time to query a temp table is the time to query a single row of the temp table multiplied by the number of rows in the table.

$$T_{query} \quad = \quad n_{tt} * (t_r \; + \; t_c * c_{tt})$$

A related quantity we model is b_{query}, the number of temp table query batches (Phase 2 batches) that are issued per temp table creation batch (Phase 1 batch). If d_{tt} is the number of dimension attributes in a temp table and n_{agg} is the number of measure attributes, the given temp table contains results for $(d_{tt} * n_{agg})$ views, and therefore $d_{tt} * n_{agg}$ queries will be made against the table. Further, since each temp table contains d_{tt} dimension attributes, there will be $Min(\frac{d}{d_{tt}}, n_{conn})$ temp tables being queried in any batch, where n_{conn} is the number of queries executing in parallel. Therefore, the total number of queries is $d_{tt} * n_{agg} * Min(\frac{d}{d_{tt}}, n_{conn})$. Finally, since

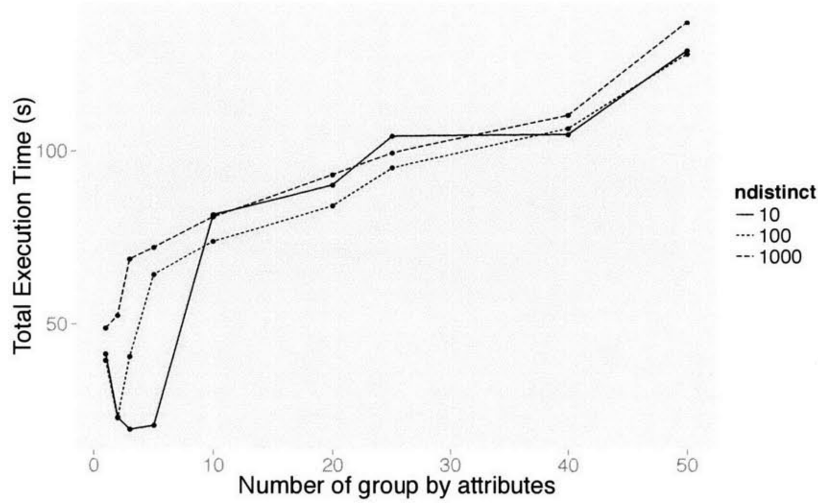


Figure 6-10: Total Execution Time vs. Number of Distinct Values

n_{conn} of these queries can execute at the same time, b_{query} can be modeled as follows:

$$b_{query} = \frac{d_{tt} * nAgg * Min(\frac{d}{d_{tt}}, n_{conn})}{n_{conn}}$$

Using the three definitions above, we can model the total time taken by SEEDB as shown in Equation 6.3. If there are b_{create} Phase 1 batches, then the total execution time is equal to the time taken to complete one batch multiplied by b_{create}. In turn, the time taken to complete a Phase 1 batch is the time taken to create temp tables and subsequently query them in Phase 2 batches. Since these two steps take place in parallel, we approximate the completion time for a Phase 1 batch as the difference between the temp table query time and temp table creation time.

$$T_{total} = \|(T_{create} - T_{query} * b_{query})\| * b_{create}$$

We evaluate the accuracy of our model by comparing the model predictions to actual performance results. Figure 6-11 shows the accuracy of our model for predicting time to create temporary tables using Equation 6.3 (Medium dataset, 1M tuples, 5 connections). Similarly, Figure 6-12 shows the accuracy of our model for predicting time to query temporary tables using Equation 6.3. Finally, Figure 6-13 shows the

Parameter	Description
d	Number of dimension attributes in table
n	Number of rows in input table
n_{tt}	Number of rows in temporary table
c_{tt}	Number of columns in the temporary table
t_r	Time to access a single row of any table
t_c	Time to process a single column in a row
t_w	Time to write a single row of a table
n_{agg}	Number of aggregate attributes computed in a single query
n_{d_i}	Number of distinct values in attribute d_i
T_{create}	Time required to create a temp table
T_{query}	Time required to query a temp table
b_{create}	Number of batches of queries required to create temp tables
b_{query}	Number of batches of queries required to query temp tables
n_{conns}	Number of queries executing in parallel

Table 6.2: Analytical model parameters

accuracy of our model for predicting total execution time as derived in Equation 6.3.

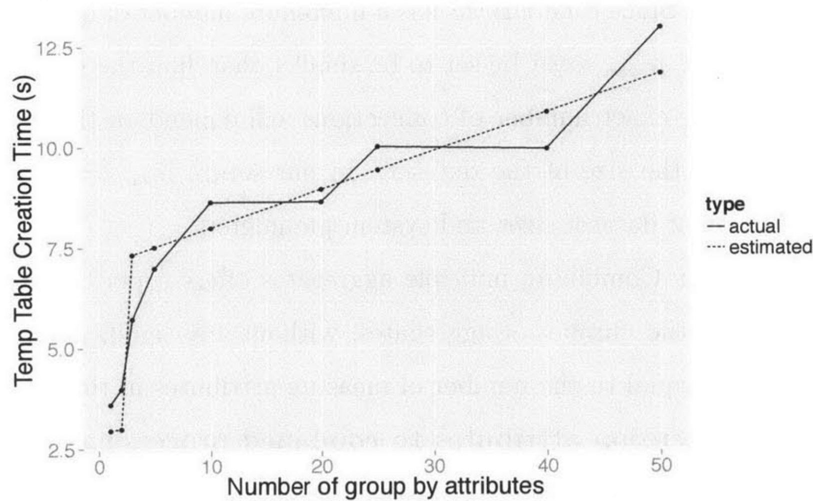


Figure 6-11: Actual and Estimated Temp Table Query Time

6.4 Choosing SeeDB parameters based on the model

Choosing Number of Parallel Queries: The speed up offered by running queries in parallel depends on the DBMS parameters such as maximum number of connections, shared buffer size, etc. In our implementation, each phase of processing can

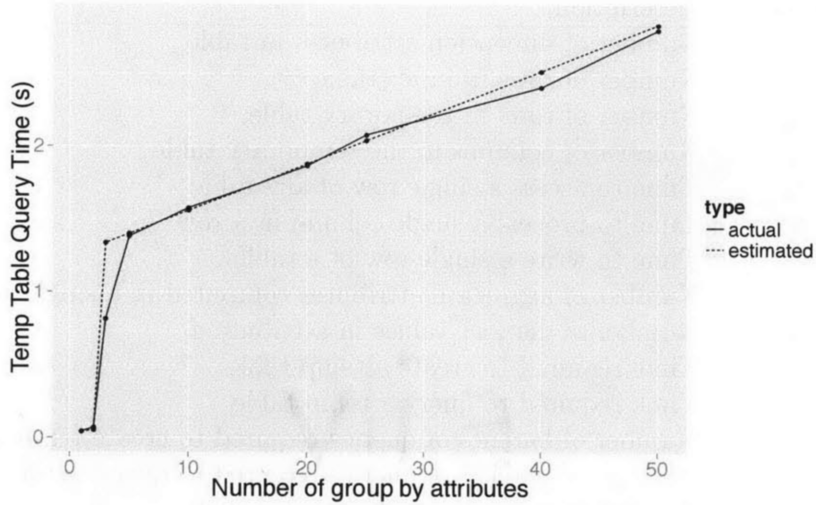


Figure 6-12: Actual and Estimated Temp Table Query Time

issue up to n_{conn} queries in parallel, so in principle, there may be up to $2*n_{conn}$ queries running in parallel. Since each DBMS has a maximum number of queries that can be executed in parallel, n_{conn} must be set to be smaller than half the maximum number of connections. The exact number of connections will depend on the other workload in the system and the size of the dataset. In our setup, n_{conns}=40 gives the best results for a range of dataset sizes and system parameters.

Choosing n_{agg}: Combining multiple aggregates offers a performance gain that is almost linear in the number of aggregates, without any significant penalty. As a result, we set n_{agg} equal to the number of measure attributes in the table.

Choosing dimension attributes to combined processing: As discussed in Section 6.2.5, the optimal number of distinct groups consistently falls in the range 10,000 to 100,000. As a result, we set n_{groups}, the maximum number of groups that any query can generate, to 100,000. This parameter is used to pick dimension attributes that will be combined into a single view query. For a set of attributes $a_1 \ldots a_n$, the maximum number of distinct groups that can be generated is $\prod_i a_i$. This is the worst case bound since correlation between two attributes can only decrease the number of distinct groups. SeeDB models the problem of grouping attributes with a constraint on number of groups as a variant of bin-packing. Specifically, SeeDB adopts the

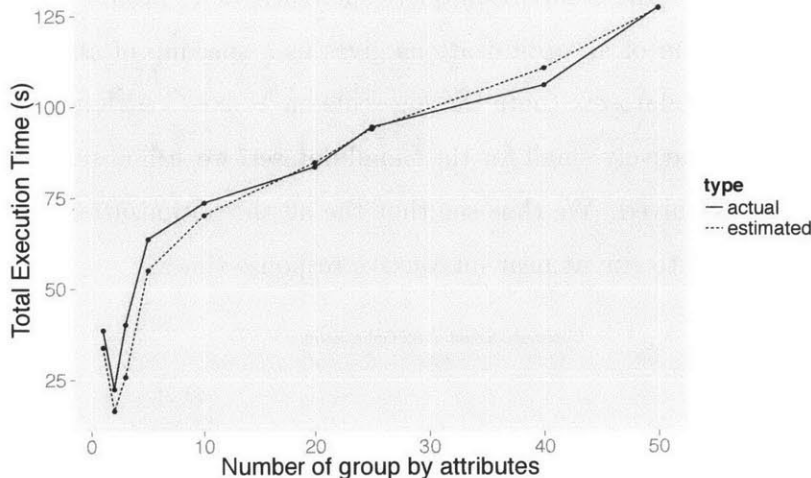


Figure 6-13: Actual and Estimated Total Execution Time

following problem formulation.

Let n_{d_i} be the number of distinct values for dimension attribute d_i. The traditional definition for bin packing states that given bins of size V and n items with size $a_1 \ldots a_n$, find the minimum integer number of bins and a corresponding partition of the set of items such that $\sum_j a_j \leq V$ for all a_j belong to any given partition. In our setting, there is a limit on the *product* (as opposed to the sum) of the sizes of items. As a result, we formulate the problem as follows: Given bins (queries) of size V (V=100,000) and d attributes with sizes $log(n_{d_i})$, find the minimum number of bins and a corresponding partition of the d attributes such that $\sum_j log(n_{d_j})$, i.e. $\prod_j n_{d_i}, \leq V$. Bin-packing has been well studied and we use off-the-shelf software [1] to perform bin-packing on the dimension attributes.

6.5 Experimental Evaluation with All Optimizations

We now show performance results for SEEDB using the optimal parameter settings described above. Specifically, we set n_{conns}=40, $n_{agg}=m$ (i.e. number of measure attributes) and n_{groups}=100,000 for bin-packing. Further, we apply the optimization

of combinging target and comparison queries for each view. From Figure 6-14, we see that the combination of all optimizations gives us a speedup of about 100X for the Medium and Large datasets (note the log scale on Y axis). Although the impact of optimizations is relatively small for the Small dataset, we still observe that the total execution time is halved. We thus see that the all the optimizations taken together can enable SeeDB to run at near-interactive response times.

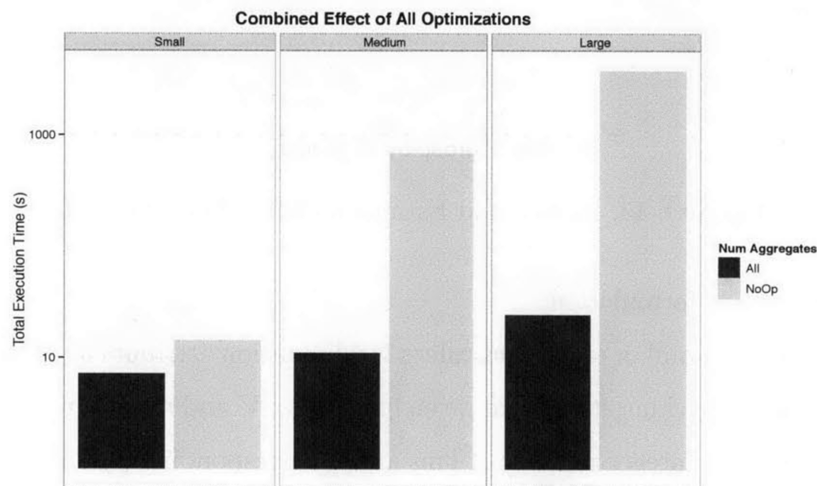


Figure 6-14: Performance Speedup With Optimizations

Chapter 7

Related Work

Recently, there has been renewed interest in the database community for building data analysis tools. As compared to previous work, the emphasis of this wave of tools is to make the data analysis accessible to non-expert users, to make the analysis process interactive, and to enable the analysis to scale to large datasets. Our work on the SEEDB system is related to existing work in several lines of research.

7.1 Interactive Data Visualization Tools

Over the past few years, the research community has introduced a number of interactive data analytics tools such as ShowMe, Polaris, and Tableau [18, 10]. Unlike SEEDB, which recommends visualizations automatically, these tools place the onus on the analyst to specify the visualization to be generated. For datasets with a large number of attributes, it is not possible for the analyst to manually study all the attributes; hence, interactive visualization needs to be augmented with automated techniques of visualization. Profiler is one such automated tool that allows analysts to detect anomalies in data [7].

Similar visualization specification tools have also been introduced by the database community, including Fusion Tables [3] and the Devise [9] toolkit.

7.2 Data Cubes

The work done in SeeDB is of a flavor similar to previous literature in building and browsing OLAP data cubes. Data cubes have been very well studied in the literature [5, 4], and work such as [14, 16, 15, 11] has explored the questions of allowing analysts to find explanations for trends, get suggest for cubes to visist, identify generalizations or patterns starting from a single cube. While we can reuse some of the similarlity metrics proposed in these papers, the exact techniques are different because of the specific problem setup.

7.3 General Purpose Data Analysis Tools

Our work is also related to work on building general purpose data analysis tools on top of databases. For example, MADLib [6] implements various analytic functions inside the database. MLBase [8] provides a platform that allows users to run various machine learning algorithms on top of the Spark system [24]. Similarly, statistical analysis packages such as R, SAS and Matlab could also be used to perform analysis similar to SeeDB.

7.4 Multi-query optimization

Since SeeDB must execute a large number of queries, there are several opportunities for performing multi-query optimization and we explore some of these strategies in Section 5 and build an analytical model of SeeDB performance. For this, we draw upon work in the areas of multi-query optimization and modeling parallel query execution [23, 17, 25].

Chapter 8

Discussion

This thesis describes our implementation of SeeDB and its evaluation. There are several ways to extend SeeDB to be more efficient, more flexible and more helpful in the data analysis process. We now describe some directions for future work

8.1 Making SeeDB more efficient

Interactive response times for SeeDB are currently achieved mainly through aggressive sampling and query optimization strategies. Another way to achieve lower response times is to perform aggressive pruning of views even before the corresponding view queries are executed by the DBMS. This pruning can be performed if we can claim with high probability that certain views are guaranteed to be less interesting than other views. We can leverage information about data types, data distributions and correlations in order to perform this pruning of views.

Another approach to making SeeDB more efficient is to choose a backend DBMS that is particularly suited for the workloads generated by SeeDB. The advantage of having SeeDB as a wrapper over the database is that we can replace backends without changes to the SeeDB code. In particular, it would be instructive to compare how databases with different data layouts can speed up SeeDB processing. One may expect that column stores like Vertica may be more efficient at processing SeeDB workloads since individual columns would be stored separately. It is however also

likely that optimization strategies would be significantly different depending on the data layout. Similarly, a comparison of column stores vs. main memory databases like VoltDB could also provide interesting insights. Finally, we can attempt to speed up workloads like SeeDB by implementing operators inside the database that can leverage shared scans for tables.

8.2 Making SeeDB more flexible

For data analysis tools to be effective, they must achieve the right balance of automation and interactivity in the analysis process. For instance, in SeeDB, merely providing the analyst the system's pick of ten most interesting views is insufficient. We must not only provide explanations for our choice but also allow the user to interrogate our views directly and further manipulate the data iteratively.

We can offer the user even more flexibility by providing a diverse set of distance metrics and allowing the user to specify the distance metric. In the future, we could also attempt to learn a distance metric based on user's feedback. Similarly, we can leverage user feedback to learn the type of views that a user finds interesting and use that model to prune uninteresting views.

8.3 Making SeeDB more helpful in data analysis

The model used by SeeDB to measure differences in data is only one of many ways to find differences in data. One can image applying a host a techniques including statistical significance testing, classification and clustering for this purpose. In the future, SeeDB should be augmented to include these additional difference-finding techniques. It is likely that this would require redefining distance metrics and completely redesigning optimizations for these operations. However, it would be possible to use the current aggregate+group-by framework used by SeeDB as a pruning step for these more advanced techniques. The addition of this functionally is not a trivial change, but it would make SeeDB much more useful for data analysts.

Chapter 9

Conclusion

In this thesis, we described a prototype of the SEEDB system used to find interesting visualizations of any database query. Given a query Q, SEEDB identifies and highlights the top-k most interesting views of Q using techniques based on deviation. We implement SEEDB as a layer on top of a relational DBMS and describe the various optimizations strategies used to provide near-interactive response times for a range of datasets. Our experimental evaluation demonstrates the benefits of each of the different optimizations. We build a model to predict the performance of SEEDB as a function of SEEDB and the DBMS parameters. We then use this model to pick optimal parameters for SEEDB over a range of diverse datasets. Finally, we demonstrate that the combination of SEEDB optimizations offers almost a 100X speedup in total execution time and allows automatic visualization to be performed in an interactive manner.

Bibliography

[1] Glpk for java, 2014. [Online; accessed 20-May-2014].

[2] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. Scuba: Diving into data at facebook. *Proc. VLDB Endow.*, 6(11):1057–1067, August 2013.

[3] Hector Gonzalez et al. Google fusion tables: web-centered data management and collaboration. In *SIGMOD Conference*, pages 1061–1066, 2010.

[4] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997.

[5] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 205–216, New York, NY, USA, 1996. ACM.

[6] Joseph M. Hellerstein, Christoper Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The madlib analytics library: Or mad skills, the sql. *Proc. VLDB Endow.*, 5(12):1700–1711, August 2012.

[7] Sean Kandel et al. Profiler: integrated statistical analysis and visualization for data quality assessment. In *AVI*, pages 547–554, 2012.

[8] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[9] Miron Livny et al. Devise: Integrated querying and visualization of large datasets. In *SIGMOD Conference*, pages 301–312, 1997.

[10] Jock D. Mackinlay et al. Show me: Automatic presentation for visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1137–1144, 2007.

[11] Carlos Ordonez and Zhibo Chen. Exploration and visualization of olap cubes with statistical tests. In *Proceedings of the ACM SIGKDD Workshop on Visual Analytics and Knowledge Discovery: Integrating Automated Analysis with Interactive Exploration*, VAKD '09, pages 46–55, New York, NY, USA, 2009. ACM.

[12] Aditya Parameswaran, Neoklis Polyzotis, and Hector Garcia-Molina. Seedb: Visualizing database queries efciently. In *VLDB*, volume 7, pages 325–328, 2013.

[13] Postgresql.org. Postgresql: Number of database connections, 2014. [Online; accessed 20-May-2014].

[14] Sunita Sarawagi. Explaining differences in multidimensional aggregates. In *VLDB*, pages 42–53, 1999.

[15] Sunita Sarawagi. User-adaptive exploration of multidimensional data. In *VLDB*, pages 307–316, 2000.

[16] Gayatri Sathe and Sunita Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, 2001.

[17] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[18] Chris Stolte et al. Polaris: a system for query, analysis, and visualization of multidimensional databases. *Commun. ACM*, 51(11):75–84, 2008.

[19] Chaoli Wang and Han-Wei Shen. Information theory in scientific visualization. *Entropy*, 13(1):254–273, 2011.

[20] Wikipedia. Jensen shannon divergence — wikipedia, the free encyclopedia, 2013. [Online; accessed 16-July-2013].

[21] Wikipedia. Kullback leibler divergence — wikipedia, the free encyclopedia, 2013. [Online; accessed 16-July-2013].

[22] Wikipedia. Statistical distance — wikipedia, the free encyclopedia, 2013. [Online; accessed 16-July-2013].

[23] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proc. VLDB Endow.*, 6(10):925–936, August 2013.

[24] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[25] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 723–734. VLDB Endowment, 2007.