SEEDB: Supporting Visual Analytics with Data-Driven Recommendations

Manasi Vartak
MIT
mvartak@mit.edu

Samuel Madden
MIT
madden@csail.mit.edu

Aditya Parameswaran
University of Illinois (UIUC)
adityagp@illinois.edu

Neoklis Polyzotis
Google
alkispolyzotis@gmail.com

ABSTRACT

Data analysts often build visualizations as the first step in their analytical workflow. However, when working with high-dimensional datasets, identifying visualizations that show relevant or desired trends in data can be laborious. We propose SEEDB, a visualization recommendation engine to facilitate fast visual analysis: given a subset of data to be studied, SEEDB intelligently explores the space of visualizations, evaluates promising visualizations for trends, and recommends those it deems most "useful" or "interesting". The two major obstacles in recommending interesting visualizations are (a) *scale*: dealing with a large number of candidate visualizations and evaluating all of them in parallel, while responding within interactive time scales, and (b) *utility*: identifying an appropriate metric for assessing interestingness of visualizations. For the former, SEEDB introduces *pruning optimizations* to quickly identify high-utility visualizations and *sharing optimizations* to maximize sharing of computation across visualizations. For the latter, as a first step, we adopt a deviation-based metric for visualization utility, while indicating how we may be able to generalize it to other factors influencing utility. We implement SEEDB as a middleware layer that can run on top of any DBMS. Our experiments show that our framework can identify interesting visualizations with high accuracy. Our optimizations lead to *multiple orders of magnitude speedup* on relational row and column stores and provide recommendations at interactive time scales. Finally, we demonstrate via a user study the effectiveness of our deviation-based utility metric and the value of recommendations in supporting visual analytics.

1. INTRODUCTION

Data visualization is often the first step in data analysis. Given a new dataset or a new question about an existing dataset, an analyst builds various visualizations to get a feel for the data, to find anomalies and outliers, and to identify patterns that might merit further investigation. However, when working with high-dimensional datasets, identifying visualizations that show interesting variations and trends in data is non-trivial: the analyst must manually specify a large number of visualizations, explore relationships between various attributes (and combinations thereof), and examine different subsets of data before finally arriving at visualizations that are interesting or insightful. This need to manually specify and examine every visualization hampers rapid analysis and exploration.

In this paper, we tackle the problem of automatically identifying and recommending visualizations for visual analysis. One of the core challenges in recommending visualizations is the fact that whether a visualization is interesting or not depends on a host of factors. In this paper, we adopt a simple criterion for judging the *interestingness* of a visualization: a visualization is likely to be interesting if it displays *large deviations from some reference* (e.g. another dataset, historical data, or the rest of the data.) While simple, we find in user studies (Section 6) that deviation can often guide users towards visualizations they find interesting. Of course, there are other elements that may make a visualization interesting. Examples include aesthetics (as explored in prior work [37, 23]), the particular attributes of the data being presented (our interactive tool allows analysts to choose attributes of interest) or other kinds of trends in data (for example, in some cases, a *lack* of deviation may be interesting.) Therefore, while our focus is on visualizations with large deviation, we develop a system, titled SEEDB, and underlying techniques that are largely agnostic to the particular definition of interestingness. In Section 7, we describe how our system can be extended to support a generalized utility metric, incorporating other criteria in addition to deviation.

Given a particular criteria for interestingness, called the *utility metric*, the goal of recommending visualizations based on this metric raises several challenges: first, even for a modest dataset with a small number of attributes, the number of visualizations that need to be considered is often in the hundreds or thousands. For some datasets, simply generating each of these visualizations can take many minutes (as we will see in this paper). Second, evaluating each of these visualizations for utility requires repeated computations on the same underlying data, wasting time and computational resources. Third, recommendations need to be made at interactive speeds, necessitating approximations that return visualizations with slightly lower accuracy. Addressing these challenges and trade-offs in our system, SEEDB, is the primary focus of this paper.

We begin with an illustrative example that explains the SEEDB use case and motivates our deviation-based utility metric.

EXAMPLE 1.1. *Consider a journalist performing research for a news article about millennials. Previous analyses show that millennials are getting married at an older age than previous generations, raising questions about how this change affects wider society. Consequently, the journalist is examining how marital-status impacts socio-economic indicators like education and income, among others. She uses the US Census data [30] to conduct her analysis comparing* unmarried *US adults to* married *US adults.*

As is common in many analytical workflows, the journalist begins by using her favorite visualization software to graph various indicators in the data. For instance, she may build a chart showing average income as a function of marital status, visualize marital status as a function of education, plot the correlation with race and gender, visualize impact on hours worked per week, and so on. Depending on the types of visualizations created, the number of possible visualizations grows exponentially with the number of indicators in the dataset. As a result, creating and examining all possible visualizations quickly becomes untenable.

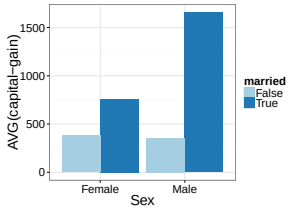
We have identified that across many analyses, visualizations that

Sex	Married	Capital Gain
Female	True	758
Female	False	380
Male	True	1657
Male	False	356

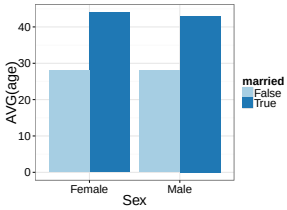
(a) Data: Avg Capital Gain vs. Sex

Sex	Married	Age
Female	True	44
Female	False	28
Male	True	43
Male	False	28

(b) Data: Avg Age vs. Sex



(c) Interesting Visualization



(d) Uninteresting Visualization

Figure 1: Motivating Example

*show trends in the target data (*unmarried *adults) that deviate from trends in reference data (*married *adults) are potentially interesting to analysts. For example, from our user study (Section 6), we know that for this particular task involving the Census dataset, analysts find the visualization in Figure 1c interesting since it depicts an aspect of the data for unmarried adults that is significantly different from the equivalent aspect for married adults. Specifically, the chart shows that although capital gain for female and male unmarried adults is approximately equal, capital gain for female, married adults is only half that of male, married adults. The same user study also shows that Figure 1d is a visualization that analysts do not find interesting. Note that this chart shows that age does* not *show different trends between unmarried and married adults.*

The example above suggests that visualizations that depict deviations from a reference are potentially interesting to analysts; our goal in the current paper is to build a system that uses deviation as a means to identify the most interesting visualizations from a large set of potential visualizations. No existing system that we are aware of makes use of variation from a reference to recommend visualizations. Current visualization packages like Spotfire [3] and Tableau [37] have limited capabilities for recommending visualizations. Their recommendations only implement rules-of-thumb regarding chart aesthetics such as choice of chart type, colors and marks. As noted previously, in addition to our deviation-based metric, there are of course many other dimensions (aesthetics, user preferences, etc.) that can influence the perceived utility of a visualization, a few of which we discuss in Section 7, leaving a detailed investigation of such metrics for future work.

Our implementation of SEEDB incorporates an end-to-end data querying and visualization environment that allows analysts to manually generate their own visualizations (like Tableau or Spotfire), or get data-driven recommendations on demand, which can be further refined using the manual interface. We chose to support both automated and manual interaction in SEEDB because we believe that a *mixed-initiative interface* [14] is essential for keeping analysts in the loop and allowing them to drive the analysis process.

In developing SEEDB as a middleware layer that can run on any database system, we develop and validate the use of two orthogonal techniques to make the problem of recommending visualizations based on deviation tractable:

- *Sharing Computation.* We develop a suite of multi-query optimization techniques to share computation among the candidate visualizations, reducing time taken by 20X.
- *Pruning Computation.* We develop pruning techniques to avoid wasting computation on obviously low-utility visualizations, adapting techniques from traditional confidence-interval-based [13] top-k ranking and multi-armed bandits [40], further reducing time taken by 5X.

Lastly, we develop a general *phase-based execution framework* that allows us to leverage the benefits of these two techniques in tandem, reducing the time for execution by over 100X and making many recommendations feasible in real-time. In summary, the contributions of this paper are:

- We build a system that uses deviation from reference as a criterion for finding the top-k most interesting visualizations for an analytical task (Section 2).
- We present the design of SEEDB as a middleware layer that can run on any SQL-compliant DBMS (Section 3).
- We describe SEEDB's execution engine (Section 4), that uses sharing techniques to share computation across visualizations (Section 4.1) and pruning techniques to avoid computation of low-utility visualizations (Section 4.2).
- We evaluate the performance of SEEDB and demonstrate that SEEDB can identify high-utility visualizations with high accuracy and at interactive time scales (Section 5).
- We present the results of a controlled user study that validates our deviation-based utility metric, and evaluates SEEDB against a manual chart construction tool showing that SEEDB can speed up identification of interesting visualizations (Section 6).
- We describe how we can extend SEEDB to capture other utility criteria beyond deviation (Section 7).

Finally, we note that the vision for SEEDB was described in a vision paper [27] and presented as a demonstration [38], but neither of these short papers described detailed, concrete algorithms or architectures, or presented any form of evaluation. Specifically, the present work builds upon the SEEDB vision by proposing a novel, general-purpose phase-based execution framework that can leverage both sharing and pruning optimizations (that were described very briefly in the short papers) and presenting both a performance study of the system as well as a extensive user study demonstrating the efficacy of our system in aiding analysis.

2. PROBLEM STATEMENT

As is standard in OLAP, and in visual analytics tools such as Tableau and Polaris [1, 37], we focus on a database D with a snowflake schema. We denote the attributes that we would like to group-by in our visualizations as *dimension attributes*, A, and the attributes that we would like to aggregate in our visualizations as *measure attributes*, M. Further, we denote by F the set of potential aggregate functions over the measure attributes (e.g. COUNT, SUM, AVG). For visualization purposes, we assume that we can group D along any of the dimension attributes A and we can aggregate any of the measure attributes M. This leads to a two-column table that can be easily visualized via standard visualization mechanisms, such as bar charts or trend lines. (Recent work has shown that bar charts are the overwhelming majority of visualizations created using visual analytics tools [25].) Our techniques also apply to the general Polaris table algebra [37], where we can aggregate across multiple attributes at once, and group-by multiple attributes, potentially leading to more than two columns. For ease of exposition, we focus on two-column result visualizations in this paper, which can be readily visualized using bar charts or trend lines.

In addition to the database D, we assume that the analyst has indicated a desire to explore a subset of data specified by a query Q. The goal of SEEDB is to recommend visualizations of Q that have high utility (which we measure using deviation, as explained below). The class of queries Q posed over D that we support encompass a general class of queries that select a horizontal fragment of the fact table and one or more dimension tables. Conceptually,

we can view this as a simple selection query over the result of joining all the tables involved in the snowflake schema. That said, we can also support projections and joins which essentially have the effect of respectively dropping certain columns or tables from consideration in the visualizations. Thus, we support a general class of select-project-join (SPJ) queries over the snowflake schema. For the purpose of this discussion, we focus on simple selection queries over the result of joining all the tables in the snowflake schema. We note that this class of queries suffices for most visualization tasks. For instance, in our illustrative example, Q can select any subset of records from the Census table. We denote the result of Q as D_Q.

Each SEEDB visualization can be translated into an aggregate / group-by query on the underlying data. We represent a visualization V_i as a function represented by a triple (a, m, f), where $m \in M, a \in A, f \in F$. We call this an *aggregate view* or simply a *view*. The aggregate view performs a group-by on a and applies the aggregation function f to measure attribute m. As an example, $V_i(D)$ represents the results of grouping the data in D by a, and then aggregating the m values using f; $V_i(D_Q)$ represents a similar visualization applied to the data in D_Q.

SEEDB determines the utility of visualizations via deviation; visualizations that show different trends in the query dataset (i.e. D_Q) compared to a reference dataset (called D_R) are said to have high utility. The reference dataset D_R may be defined as the entire underlying dataset (D), the complement of D_Q (D - D_Q) or data selected by any arbitrary query Q' ($D_{Q'}$). The analyst has the option of specifying D_R; we use $D_R = D$ as the default if the analyst does not specify a reference. Given a view V_i, the deviation-based utility of V_i is computed as the deviation between the results of applying V_i to the query data, D_Q, and applying V_i to the reference data, D_R. View V_i applied to the results of Q can be expressed as query Q_T below. We call this the *target view*.

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

Similarly, view V_i applied to the reference data $V_i(D_R)$ can be expressed as Q_R. We call this the *reference view*.

$$Q_R = \texttt{SELECT}\ a, f(m)\ \texttt{FROM}\ D_R\ \texttt{GROUP BY}\ a$$

The (two) SQL queries corresponding to each view are referred to as *view queries*. The results of the above view queries are summaries with two columns, namely a and $f(m)$. To ensure that all aggregate summaries have the same scale, we normalize each summary into a probability distribution (i.e. the values of $f(m)$ sum to 1). For our example visualization of *Average Capital Gain vs. Sex* (Figure 1), the probability distribution for the target view $V_i(D_Q)$ (*unmarried* adults), denoted as $P[V_i(D_Q)]$ is: (F: 0.52, M: 0.48) while that for the reference view $V_i(D_R)$ (*married* adults), denoted as $P[V_i(D_R)]$ is: (F: 0.31, M: 0.69). In contrast, the distributions for the visualization *Average Age vs. Sex* are (F: 0.5, M: 0.5) and (F: 0.51, M: 0.49) for the target and reference view respectively. Qualitatively, we see that the distributions show a large deviation for the former visualization and hardly any deviation for the latter.

Given an aggregate view V_i and probability distributions for the target view ($P[V_i(D_Q)]$) and reference view ($P[V_i(D_R)]$), we define the *utility* of V_i as the distance between these two probability distributions. The higher the distance between the two distributions, the more likely the visualization is to be interesting and therefore higher the utility. Formally, if S is a distance function,

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

Computing distance between probability distributions has been well studied in the literature, and SEEDB supports a variety of distance functions to compute utility, including Earth Movers Distance, Euclidean Distance, Kullback-Leibler Divergence (K-L divergence), and Jenson-Shannon Distance. Our experiments use Earth Movers Distance as the default distance function, but in Section 7 we discuss results for other distance functions. Also in Section 7, we describe how our utility metric can be generalized to capture other aspects of interest to analysts (beyond deviation).

We can formally state the SEEDB problem as follows:

PROBLEM 2.1. *Given a user-specified query Q on a database D, a reference dataset D_R, a utility function U as defined above, and a positive integer k, find k aggregate views $V \equiv (a, m, f)$ that have the largest values of $U(V)$ among all the views (a, m, f), while minimizing total computation time.*

3. SEEDB FRONT-END & ARCHITECTURE

We now describe SEEDB's front-end user experience, and then describe the architecture and the execution engine in more detail.

Front-end Experience. SEEDB's visualization recommendation capabilities are packaged into an end-to-end visual analytics environment, with basic visualization functionalities such as those provided by Tableau. Figure 3 shows the web front-end for SEEDB comprising four parts (A) dataset selector used to connect to a database and query builder used to formulate queries; (B) visualization builder used to manually specify visualizations; (C) visualization display pane; and (D) a recommendations plugin that displays recommended visualizations. The recommendations provided by SEEDB change in response to changes in the query (B) issued against the database.

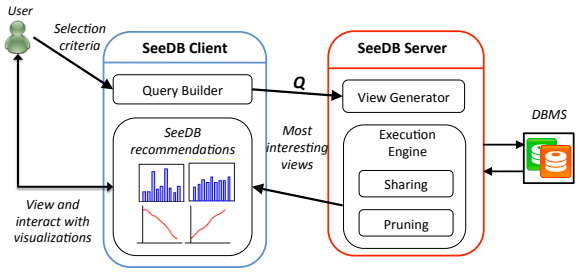


Figure 2: SeeDB Architecture

Architectural Overview. SEEDB is implemented as a middleware layer that can run on top of any SQL-compliant database system. Figure 2 depicts the overall architecture of our system. The SEEDB client is a web-based front-end that captures user input and renders visualizations produced by the SEEDB server. The SEEDB server is composed of two main components. The first component, the *view generator*, is responsible for parsing the input query, querying system metadata and generating the list of visualization queries that must be evaluated. The goal of the *execution engine* is to evaluate the collection of queries using our optimizations on top of the underlying DBMS. The selected aggregate views (i.e., those with high deviation) are sent to the SEEDB client and are displayed as visualization recommendations to the user, who can then interact with these visualizations.

Basic Execution Engine. To motivate the need for optimizations, we first describe how our execution engine would work without optimizations. To identify the k best aggregate views, SEEDB needs to do the following: For each aggregate view, it generates a SQL query corresponding to the target and reference view, and issues the two queries to the underlying DBMS. It repeats this process for each aggregate view. As the results are received, it computes the distance between the target and reference view distributions, and identifies the k visualizations with highest utility.

This basic implementation has many inefficiencies. In a table with a dimensions, m measures, and f aggregation functions, $2 \times f \times a \times m$ queries must be executed. As we show in Section 5, this can take >100s for large data sets. Such latencies are unacceptable for interactive use.

Execution Engine with Optimizations. To reduce latency in evaluating the collection of aggregate views, the execution engine applies two kinds of optimizations: *sharing*, where aggregate view queries are combined to share computation as much as possible, and *pruning*, where aggregate view queries corresponding to low utility visualizations are dropped from consideration without scanning the whole dataset. These optimizations are largely orthogonal to each other. To derive benefits from both these kinds of optimizations, we develop a *phased execution framework*. Each phase operates on a subset of the dataset. Phase i of n operates on the ith of n equally-sized partitions of the dataset. (Empirically, we have found $n = 10$ to work well, though our results are not very sensitive to the value of n.) For instance, if we have $100,000$ records and 10 phases, the $i = 4$th phase processes records $30,001$ to $40,000$. The execution engine begins with the entire set of aggregate views under consideration.

- During phase i, the SEEDB updates partial results for the views still under consideration using the ith fraction of the dataset. The execution engine applies *sharing-based optimizations* to minimize scans on this ith fraction of the dataset.
- At the end of phase i, the execution engine uses *pruning-based optimizations* to determine which aggregate views to discard. The partial results of each aggregate view on the fractions from 1 through i are used to estimate the quality of each view, and the views with low utility are discarded.

The retained aggregate views are then processed on the $i + 1$th round, and the process continues. In this manner, the set of views under consideration decreases across these phases, with all aggregate views at the start of the first phase, and only the k views left at the end of the nth phase.

Algorithm 1 Phase-based Execution Framework

```
1: viewsInRunning ← {all views}
2: for currPhase ← 1 . . . n do
3:     updateResults(viewsInRunning)
4:     pruneViews(viewsInRunning)
5: return viewsInRunning.getTopK()
```

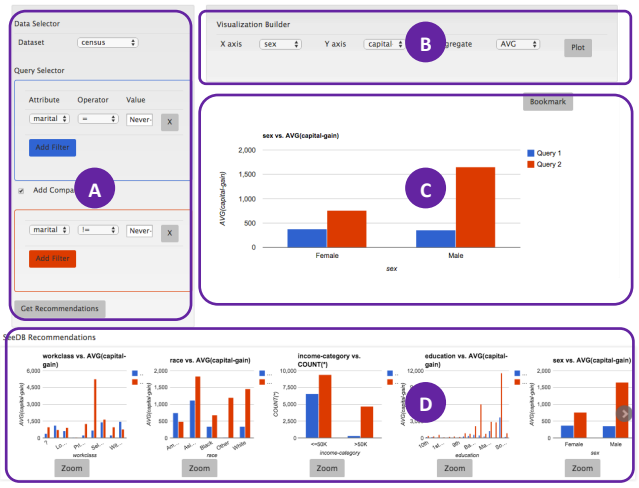


Figure 3: SEEDB *Frontend*

4. SEEDB EXECUTION ENGINE

In this section, we describe the sharing and pruning optimizations designed to optimize the generation of visualizations.

4.1 Sharing-based Optimizations

In the naive implementation, each visualization is translated into view queries that get executed independently on the DBMS. However, for a particular user input, the queries evaluated by SEEDB are very similar: they scan the same underlying data and differ only in the attributes used for grouping and aggregation. This presents opportunities to intelligently merge and batch queries, reducing the number of queries issued to the database and, in turn, minimizing the number of scans of the underlying data. Sharing computation in our setting is a special case of the general problem of multi-query optimization [35]; we discuss the relationship in more detail in Section 8. Specifically, we apply the following optimizations:

Combine Multiple Aggregates: Aggregate view queries with the same group-by attribute can be rewritten as a single query with multiple aggregations. Therefore, instead of executing (two queries each for) views $(a_1, m_1, f_1), (a_1, m_2, f_2) \ldots (a_1, m_k, f_k)$, SEEDB combines them into a single view $(a_1, \{m_1, m_2 \ldots m_k\}, \{f_1, f_2 \ldots f_k\})$ which can be executed via just two queries. We have found that there is minimal to no impact on latency for combining aggregates in both row and column stores.

Combine Multiple GROUP BYs: After applying our multiple aggregates optimization, SEEDB is left with a number of queries with multiple aggregations but only single-attribute groupings. These queries can be further combined to take advantage of multi-attribute grouping. However, unlike combining multiple aggregates, the addition of a grouping attribute can dramatically increase the number of groups that must be maintained and (possibly) lead to slower overall performance for large number of groups.

We claim (and verify in Section 5) that grouping can benefit performance so long as memory utilization for grouping stays under a threshold. Memory utilization is, in turn, proportional to the number of distinct groups present in a query. If a set of attributes $a_1 \ldots a_m$ are used for grouping, the upperbound on the number of distinct groups is given by $\prod_{i=1}^{m} |a_i|$. Given a memory budget $\mathcal{S}$, the challenge is now to determine the optimal grouping of attributes that each respect the memory budget. Formally, the problem is:

PROBLEM 4.1 (OPTIMAL GROUPING). *Given memory budget $\mathcal{S}$ and a set of dimension attributes $A = \{a_1 \ldots a_n\}$, divide the dimension attributes in A into groups $A_1, \ldots, A_l$ (where $A_i \subseteq A$ and $\bigcup A_i = A$) such that if a query Q groups the table by any A_i, the memory utilization for Q does not exceed $\mathcal{S}$.*

Notice that the above problem is isomorphic to the NP-Hard *bin-packing* problem [22]. If we let each dimension attribute a_i correspond to an item in the bin-packing problem with weight $\log(|a_i|)$, and set the bin size to be $\log \mathcal{S}$, then packing items into bins is identical to finding groups $A_1, \ldots, A_l$, such that the estimated size of any query result is below $\mathcal{S}$. We use the standard first-fit algorithm [17] to find the optimal grouping of dimension attributes.

Combine target and reference view query: Since the target and reference views differ only in the subset of data the query is executed on, SEEDB rewrites these two view queries as one. For instance, if the target and reference view queries are $Q1$ and $Q2$ respectively, they can be combined into a single query $Q3$.

$Q1$ = `SELECT` $a, f(m)$ `FROM` D `WHERE x <` 10 `GROUP BY` a

$Q2$ = `SELECT` $a, f(m)$ `FROM` D `GROUP BY` a

$Q3$ = `SELECT` $a, f(m)$, `CASE IF x <` 10 `THEN 1 ELSE 0 END` $as\ g1$, 1 $as\ g2$ `FROM` D `GROUP BY` a, $g1$, $g2$

Parallel Query Execution: SEEDB executes multiple view queries in parallel: these queries can often share buffer pool pages, reducing disk access times. However, the precise number of parallel queries needs to be tuned taking into account buffer pool contention, locking, and cache line contention, among other factors [28].

Other Optimizations: To further speedup processing, SEEDB can also pre-compute results for static views (e.g. reference views on full tables) or operate on pre-computed data samples. Such optimizations are orthogonal to the problem of efficiently evaluating a large number of views, which we must address even in the presence of pre-computation or sampling.

4.2 Pruning-Based Optimizations

In practice, most visualizations are low-utility, meaning computing them wastes computational resources. Thus, as described earlier, at the end of every phase, the execution engine uses pruning optimizations to determine which aggregate views to discard. Specifically, partial results for each view based on the data processed so far are used to estimate utility and views with low utility are discarded. The SEEDB execution engine supports two pruning schemes. The first uses confidence-interval techniques to bound utilities of views, while the second uses multi-armed bandit allocation strategies to find top utility views.

Confidence Interval-Based Pruning. Our first pruning scheme uses worst-case statistical confidence intervals to bound view utilities. This technique is similar to top-k based pruning algorithms developed in other contexts [15, 29]. Our scheme works as follows: during each phase, we keep an estimate of the mean utility for every aggregate view V_i and a confidence interval around that mean. At the end of a phase, we use the following rule to prune low-utility views: *If the upper bound of the utility of view V_i is less than the lower bound of the utility of k or more views, then V_i is discarded.* To illustrate, suppose a dataset has 4 views $V_1 \ldots V_4$ and we want to identify the top-2 views. Further suppose that at the end of phase i, V_1-V_4 have confidence intervals as shown in Figure 4. Views V_1 and V_2 have the highest utility estimates so far and are likely to be in the top-2 views. View V_3 is currently not the top-2, but its confidence interval overlaps with that of the top-2, making it possible that V_3 could replace V_1 or V_2. The confidence interval for V_4, on the other hand, lies entirely below the confidence intervals of V_1 and V_2. Since we can claim with high probability that the utility of V_4 lies within its confidence interval, it follows that, with high probability, V_4's utility will be lower than that of both V_1 and V_2, and it will not appear in the top-2 views. We state the algorithm formally in Algorithm 2.

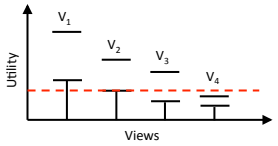


Figure 4: Confidence Interval based Pruning

Algorithm 2 Confidence Interval Based Pruning

```
1: viewsInRunning.sortByUpperbound()
2: topViews ← viewsInRunning.getTopK()
3: lowestLowerbound ← min(lowerbound(topViews))
4: for view ∉ topViews do
5:     if view.upperbound < lowestLowerbound then
6:         viewsInRunning.remove(view)
```

We use *worst case* confidence intervals as derived from the Hoeffding-Serfling inequality [36]. The inequality states that if we are given N values $y_1, \ldots, y_N$ in $[0, 1]$ with average μ, and we have have drawn m values without replacement, $Y_1, \ldots, Y_m$, then we can calculate a running confidence interval around the current mean of the m values such that the actual mean of the N is always within this confidence interval with a probability of $1 - \delta$:

THEOREM 4.1. *Fix any $\delta > 0$. For $1 \leq m \leq N - 1$, define*

$$\varepsilon_m = \sqrt{\frac{(1 - \frac{m-1}{N})(2\log\log(m) + \log(\pi^2/3\delta))}{2m}}.$$

Then: $$\Pr\left[\exists m, 1 \leq m \leq N : \left|\frac{\sum_{i=1}^{m} Y_i}{m} - \mu\right| > \varepsilon_m\right] \leq \delta.$$

In our setting, each Y_i corresponds to the an estimate of utility computed based on the records seen so far.

Multi-Armed Bandit Pruning. Our second pruning scheme employs a Multi-Armed Bandit strategy (MAB) [40][40, 5, 21]. In MAB, an online algorithm repeatedly chooses from a set of alternatives (arms) over a sequence of trials to maximize reward. We note that this is the first time that bandit strategies have been applied to the problem of identifying interesting visualizations.

A recently-studied variation of MAB focuses on finding the arms with the highest mean reward [6, 4]. This variation is identical to the problem addressed by SEEDB: our goal is find the visualizations (arms) with the highest utility (reward). Specifically, we adapt the Successive Accepts and Rejects algorithm from [6] to find arms with the highest mean reward (See Algorithm 3). At the end of every phase (Section 3), views that are still under consideration are ranked in order of their utility means. We then compute two differences between the utility means: Δ_1 is the difference between the highest mean and the $k+1$st highest mean, and Δ_n is the difference between the lowest mean and the kth highest mean. If Δ_1 is greater than Δ_n, the view with the highest mean is "accepted" as being part of the top-k (and it no longer participates in pruning computations). On the other hand, if Δ_n is higher, the view with the lowest mean is discarded from the set of views in the running. [6] proves that under certain assumptions about reward distributions, the above technique identifies the top-k arms with high probability.

Algorithm 3 MAB Based Pruning

```
1: viewsInRunning.sortByUtilityMean()
2: {ū_i} ← sorted utility means
3: Δ_1 ← ū_1 - ū_{k+1}
4: Δ_n ← ū_k - ū_n
5: if Δ_1 > Δ_n then
6:     viewsInRunning.acceptTop()
7: else
8:     viewsInRunning.discardBottom()
```

Consistent Distance Functions. Note that the two pruning schemes described above have guarantees in other settings that do not directly carry over to our setting—for example, the MAB setting assumes that each trial samples from a fixed underlying distribution, while in fact, our trials correspond to random values across m distributions (groups), which are aggregated together to form a utility estimate for a given view. In our evaluation, we show that in spite of this limitation, the pruning schemes work rather well in practice.

We can, however, get a weaker guarantee: we can show that as we sample more and more, the estimated utility $\hat{U}$ can be made to be arbitrarily close to U for all aggregate views. Essentially, this means that any pruning algorithm (including Confidence and MAB) that uses a sufficiently large sample will prune away low

utility views with high probability. We can state our claim formally in the following lemma.

LEMMA 4.1 (CONSISTENCY). *Let the target and reference visualizations both have m groups. Let $\hat{U}$ denote our estimate of the utility U based on a uniformly random sample across all m groups. Then, as the number of samples tends to ∞, $\hat{U} \to U$ with probability $1-\delta$, for as small δ as needed.*

For the purpose of the proof of the lemma above, we focus on the case where the visualization V_i corresponds to the AVG aggregate. Similar results can be shown for the SUM and STD aggregates. Unfortunately, MAX and MIN are not amenable to sampling-based optimizations, as is traditionally well-known in the approximate query processing literature [7, 16].

Additionally, we focus on the case when S is defined to be ℓ_2, i.e., the Euclidean distance metric. Once again, similar results can be shown for other distance metrics, such as the ℓ_1, the Earth Movers Distance metric, or the Jenson-Shannon metric.

We reproduce the utility equation here: $U(V_i) = S(P[V_i(D_Q)], P[V_i(D_R)])$. Here, P refers to the probability distribution of either the target visualization or the comparison visualization. P is represented as a normalized vector whose entries sum up to 1.

PROOF. (Sketch) Let us say the estimated average for the target visualization for each of the groups is $\hat{t}_i$, and the estimated averages for the comparison visualization for each of the groups is $\hat{c}_i$. We further define $\hat{t}$ (respectively $\hat{c}$) to be the estimated sum of averages for the target visualization (respectively comparison visualization). We let the true values for each of these quantities be the same variables without the hats. Then, it can be shown that that U evaluates to:

$$\hat{U} = \frac{\sum \hat{c}_i^2}{\hat{c}^2} + \frac{\sum \hat{t}_i^2}{\hat{t}^2} - 2\frac{\sum \hat{t}_i \hat{c}_i}{\hat{c}\hat{t}}$$

Now we informally describe the steps of our proof: say we sample enough to get $\hat{c}$ within ϵ of c, with a high enough probability, and we sample enough to get $\hat{t}$ within ϵ of t, with a high enough probability. Then, we have

$$\begin{aligned}\hat{U} &\geq \frac{\sum \hat{c}_i^2}{(c+\epsilon)^2} + \frac{\sum \hat{t}_i^2}{(t+\epsilon)^2} - 2\frac{\sum \hat{t}_i \hat{c}_i}{(c-\epsilon)(t-\epsilon)} \\ &\geq \frac{\sum \hat{c}_i^2}{c^2}(1-\epsilon) + \frac{\sum \hat{t}_i^2}{t^2}(1-\epsilon) - 2\frac{\sum \hat{t}_i \hat{c}_i}{(c-\epsilon)(t-\epsilon)} \\ &\geq \frac{\sum \hat{c}_i^2}{c^2}(1-\epsilon) + \frac{\sum \hat{t}_i^2}{t^2}(1-\epsilon) - 2\frac{\sum \hat{t}_i \hat{c}_i}{ct}(1+\epsilon^2+\epsilon)\end{aligned}$$

Similarly, if we have sampled enough to get the $\hat{c}_i$ and the $\hat{t}_i$ within γ close of their actual values, we will have:

$$\begin{aligned}\hat{U} \geq &\frac{\sum c_i^2}{c^2}(1+f(\gamma))(1-\epsilon) + \frac{\sum t_i^2}{t^2}(1+f(\gamma))(1-\epsilon) \\ &- 2\frac{\sum t_i c_i}{ct}(1+h(\gamma))(1+\epsilon^2+\epsilon)\end{aligned}$$

where $f(.)$ and $h(.)$ are small polynomial functions. Thus, we will have sandwiched $\hat{U}$ from the bottom by $U-\rho$, and similarly by $U+\rho'$ from the top. ρ, ρ' will be polynomials that depend on ϵ and γ. Now, we will use the Hoeffding's inequality for the last step of the proof. Hoeffding's inequality, when applied to a collection of n i.i.d. random variables, whose sum is represented by X, gives us:

$$Pr(|X - E[X]| \geq t) \leq 2e^{-\frac{2nt^2}{c^2}} \quad (1)$$

where c is a bound on the range. If we set the right hand side to some δ, and set $t = \epsilon$, we have

$$\epsilon = \sqrt{\frac{1}{2n}\ln\frac{2}{\delta}}$$

and therefore, as n tends to ∞, ϵ tends to 0, for fixed values of δ. The same holds true for $t = \gamma$. Thus, $\hat{U}$ will tend to U as the number of samples increases to infinity.

It is in fact also possible to explicitly derive a number of samples such that $\hat{U}$ is close to U within a certain error bound and with a certain probability. □

We call distance functions that have this property as *consistent distance functions*. Consistent distance functions allow pruning schemes to gather increasingly better estimates of utility values over time (as long as the samples are large enough). Specifically, we show empirically in Section 5.4 that the Confidence Interval and MAB-based pruning schemes work well for EMD (i.e., have high accuracy while reducing latency), and then in Section 7, we show that these schemes work well for other consistent distance functions.

4.3 Offline Pruning.

Even before any queries are issued to SEEDB, we have the ability to identify clusters of attributes that are strongly correlated with each other, such that if we elect to display a visualization on one of them, we can avoid displaying visualizations on others (since they would be redundant). Consider for example a flight dataset that stores names of airports as well as the corresponding airport codes. Since names of airports and airport codes have a 1:1 relationship, generating, say, average delay by airport name, and average delay by airport code would lead to identical visualizations. Consequently, it would suffice to compute and recommend only one of these visualizations.

We adopt the following steps to prune redundant views: (1) For each table, we first determine the entire space of aggregate views. (2) Next, we prune all aggregate views containing attributes with 0 or low variance since corresponding visualizations are unlikely to be interesting. (3) For each remaining view V_i, we compute the distribution $P[V_i(D)]$ for reference views on the entire dataset D. (4) The resulting distributions are then clustered based on pairwise correlation. (5) From each cluster, we pick one view to compute as a cluster representative and store "stubs" of clustered views for subsequent use. At run time, the view generator accesses previously generated view stubs, removes redundant views and passes the remaining stubs to the execution engine. Offline pruning allows us to significantly reduce the number of views (and execution time) in real datasets; for the DIAB dataset described in Section 5, we reduce the number of possible views from 72 to 41 (45% reduction), and from 70 to 54 (25% reduction) for the BANK dataset.

5. PERFORMANCE EVALUATION

In the next two sections, we present an evaluation of SEEDB both in terms of performance when returning visualizations and in terms of user studies. In both sections, we report results for SEEDB on a variety of real and synthetic datasets listed in Table 1.

In this section, we focus on performance studies, where our goal is to evaluate how well our sharing and pruning optimizations improve latency, and how our pruning optimizations affect accuracy. In each experiment, our primary evaluation metric is latency, i.e., how long does it take SEEDB to return the top-k visualizations. For experiments involving our pruning strategies, we measure quality of results through two additional metrics, namely *accuracy* and *utility distance* (discussed further in Section 5.4). Since we expect data layout to impact the efficacy of our optimizations, we evaluate our techniques on Postgres, a row-oriented database (denoted ROW) as well as Vertica, a column-oriented database (denoted COL).

The following experiments use *earth mover distance (EMD)* as

Name	Description	Size	|A|	|M|	Views	Size (MB)
Synthethic Datasets						
SYN	Randomly distributed, varying # distinct values	1M	50	20	1000	411
SYN*-10	Randomly distributed, 10 distinct values/dim	1M	20	1	20	21
SYN*-100	Randomly distributed, 100 distinct values/dim	1M	20	1	20	21
Real Datasets						
BANK	Customer Loan dataset	40K	11	7	77	6.7
DIAB	Hospital data about diabetic patients	100K	11	8	88	23
AIR	Airline delays dataset	6M	12	9	108	974
AIR10	Airline dataset scaled 10X	60M	12	9	108	9737
Real Datasets - User Study						
CENSUS	Census data	21K	10	4	40	2.7
HOUSING	Housing prices	0.5K	4	10	40	<1
MOVIES	Movie sales	1K	8	8	64	1.2

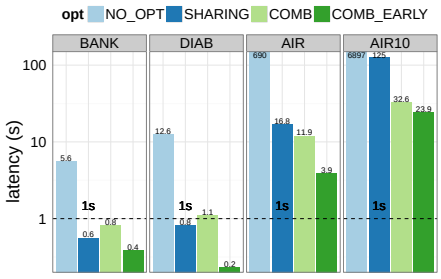
Table 1: Datasets used for testing

our distance function for computing deviation (Section 2). In Section 7, we briefly discuss results of using different distance functions to compute deviation. All experiments were run on a single machine with 8 GB RAM and a 16 core Intel Xeon E5530 processor. Unless described otherwise, experiments were repeated 3 times and the measurements were averaged.

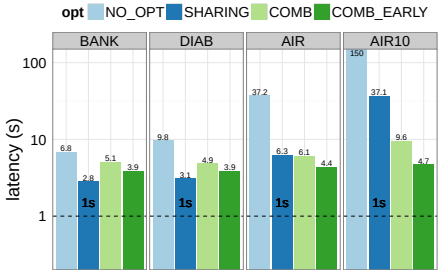
We begin by presenting a summary of our experimental findings and then dive into performance results for individual optimizations.

5.1 Summary of Findings

Figures 5.a and 5.b show a summary of SEEDB performance for the four (large) real datasets from Table 1 (BANK, DIAB, AIR and AIR10). For each dataset, we show the latencies obtained on the ROW and COL store by the basic SEEDB framework (NO_OPT), by our sharing optimizations (SHARING), and by the combination of our sharing and pruning optimizations (COMB). We also show latencies for early result generation with COMB (COMB_EARLY), where we return approximate results as soon as the top-k visualizations have been identified. The results in Figure 5 use the CI pruning scheme and k=10.



(a) Optimization results for ROW

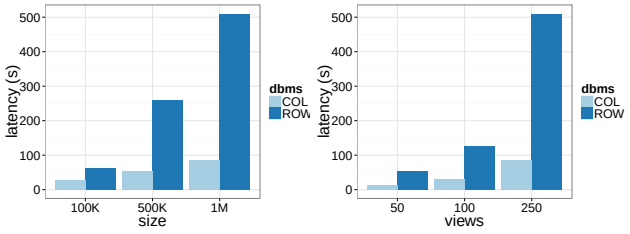


(b) Optimization results for COL

Figure 5: Performance gains from all optimizations

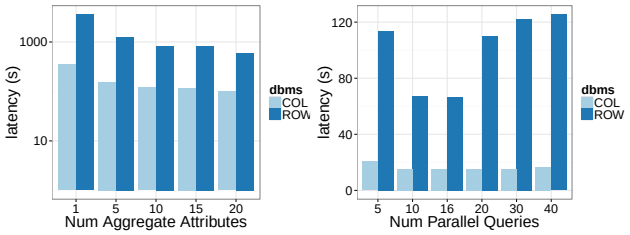
- [*> 100X speedup overall*] The combination of our sharing and pruning optimizations provides a speedup of up between 50X (COMB) – 300X (COMB_EARLY) for ROW (Figure 5a) and 10X (COMB) – 30X (COMB_EARLY) for COL (Figure 5b). This reduces latencies for small datasets like DIAB from 12s to 200ms, and from almost 2 hrs to tens of seconds for large datasets like AIR10.
- [*8–20X speedup from sharing*] The sharing optimizations (Section 4.1) alone produce performance gains of up to 20X for ROW and 8X for COL.
- [*5X speedup from pruning without loss of accuracy*] Pruning optimizations (Section 4.2) provide additional gains of up to 5X. Early result return, in particular, enables real-time response for large datasets, e.g. for AIR, the COMB_EARLY strategy allows SEEDB to return results in under 4s while processing the full dataset takes tens of seconds. We also find that quality of results is not adversely affected by pruning: the utility distance (defined later) for our pruning strategies is close to 0.
- [*Multiplicative gains*] A gain of 20X from sharing optimizations in the ROW store combines with the 5X gain from pruning to produce an overall gain of over 100X (Figure 5a).
- [*Gains improve on larger datasets*] The overall gain is much larger for AIR10 (300X) vs. BANK (10X). We find that our SHARING optimization is best suited for small datasets like BANK and DIAB, while COMB and COMB_EARLY are essential for large datasets like AIR and AIR10.

In the next sections, we discuss the performance of individual optimizations and how they relate to the overall performance gain.



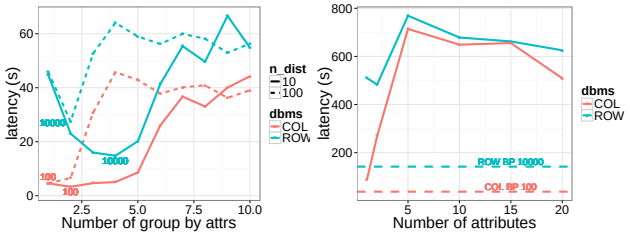
(a) Latency vs. Table size *(b) Latency vs. Num Views*

Figure 6: Baseline performance



(a) Latency vs. aggregates *(b) Effect of parallelism*

Figure 7: Effect of Group-by and Parallelism



(a) Latency vs. Num of Groups *(b) Latency vs. Num Dimensions*

Figure 8: Effect of Groups and Dimensions

5.2 Basic SEEDB Framework

Summary: Applying no optimizations leads to latencies in the 100s of seconds for both ROW and COL; latency increases linearly in the size of the dataset and number of views. Without any optimizations, the basic SEEDB serially executes two SQL queries for

each possible view. Figure 6a shows latency of SEEDB vs. the number of rows (100K rows–1M rows) in the dataset, while Figure 6b shows latency as a function of the number of views (50–250). These charts show results for the SYN dataset obtained by varying the size of the table and number of attributes (SYN is comparable to the AIR dataset). First, notice that the basic framework with no optimizations has very poor performance: latency for ROW is between 50-500s, while it is between 10-100s for COL. This is because, depending on the dataset, both ROW and COL run between 50 to 250 SQL queries for each SEEDB invocation. Second, COL runs about 5X faster than ROW. This is expected because most queries only select a few attributes, benefitting column-stores. Third, as expected, the latency of the basic framework is proportional to the number of rows as well as the number of views in the table. Since the latencies for the basic framework are very high for interactive applications, it is clear that aggressive optimization needs to be employed.

5.3 Sharing Optimizations

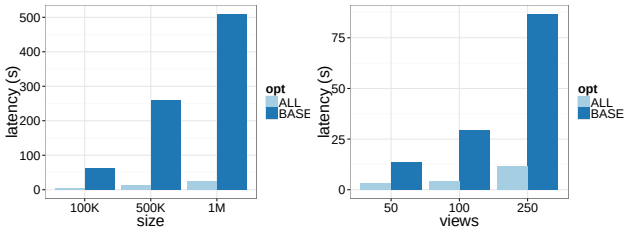
In this section, we study the performance of the sharing optimizations described in Section 4.1. Recall that the goal of these optimizations is to reduce the number of queries run against the DBMS and to share scans as much as possible between queries. The following experiments report results on the synthetic datasets SYN and SYN* (Table 1). We chose to test these optimizations on synthetic data since we can control all parameters of the data including size, number of attributes, and data distribution. (Results on real datasets are shown in Figure 5a and 5b).

Combining Multiple Aggregates: *Summary: Combining view queries with the same group-by attribute but different aggregates gives a 3-4X speedup for both row and column stores.* To study the limits of adding multiple aggregates to a single view query, we varied the maximum number of aggregates in any SEEDB-generated SQL query (n_{agg}) between 1 and 20. The resulting latencies on the SYN dataset are shown in Figure 7a (log scale on the y-axis). As we can see, latency reduces consistently with the number of aggregations performed per query. However, the latency reduction is not linear in n_{agg} because larger n_{agg} values require maintenance of more state and access more columns in a column store. Overall, this optimization provides a 4X speedup for ROW and 3X for COL.

Parallel Query Execution: *Summary: Running view queries in parallel can offer significant performance gains.* Executing SEEDB-generated SQL queries in parallel can provide significant performance gains because queries can share buffer pool pages. However, a high degree of parallelism can degrade performance for a variety of reasons [28]. Figure 7b shows how latency varied as we varied the number of parallel SQL queries issued by SEEDB. As expected, low levels of parallelism produced sizable performance gains but high levels led to degraded performance. The optimal number of queries to run in parallel is approximately 16 (equal to the number of cores), suggesting that choosing a degree of parallelism equal to the number of cores is a reasonable policy.

Combining Multiple Group-bys: *Summary: Combining multiple view queries each with a single group-by attribute into a single query with multiple grouping improves performance by 2.5X in row stores.* We now study the effect of combining multiple queries each with a single group-by into one query with multiple grouping. Unlike the multiple aggregates optimization, the impact of combining group-bys is unclear due to the significantly larger memory requirement as well as post-processing cost. We claim in Section 4.1 that grouping can benefit performance so long as the total memory utilization stays under a threshold. To verify our claim, we ran an experiment with datasets SYN*-10 and SYN*-100. For each dataset, we varied the number of group-by attributes in SEEDB-generated SQL queries (n_{gb}) between 1 and 10. Since each attribute in SYN*-10 has 10 distinct values and that in SYN*-100 has 100, a query with $n_{gb} = p$ will require memory proportional to $\max(10^p$, num_rows) for SYN*-10 and proportional to $\max(100^p$, num_rows) for SYN*-100. The results of the experiment are shown in Figure 8a. We see that as the number of group-by attributes increases from 1 to 10, the latency of ROW (blue) decreases initially. However, once the memory budget $\mathcal{S}_{ROW}$ (proxied by the number of distinct groups) exceeds 10000, latency increases significantly. We see a similar trend for COL, but with a memory budget $\mathcal{S}_{COL}$ of 100.[1] Thus, we find empirically that memory usage from grouping is in fact related to latency and that optimal groupings must keep memory usage under a threshold.

To evaluate the gains offered by our bin-packing optimization, we evaluate two methods to perform grouping, MAX_GB and BP. MAX_GB simply sets a limit on the number of group-bys in each query (n_{gb}) where as BP applies our bin-packing strategy using the respective memory budgets. Figure 8b shows a comparison of the two methods. To evaluate MAX_GB, we varied n_{gb} was varied between 1 and 20 (solid lines). Since SYN contains attributes with between 1 – 1000 distinct values, memory utilization for a given n_{gb} can be variable. For example, n_{gb} = 3 can have anywhere between 1 and 10^9 distinct groupings, thus breaking the memory budget for some groupings. Because groupings with MAX_GB depends on order, results in Figure 8b are averages over 20 runs. The dotted lines show the latency obtained by performing optimal grouping via bin-packing. Unlike MAX_GB, BP consistently keeps memory utilization under the memory budget. Consequently, we observe that BP improves performance for both ROW and COL. We observe a significant, 2.5X improvement in ROW because the large memory budget $\mathcal{S}_{ROW}$ allows many queries to be combined. COL, in contrast, shows a much less pronounced speedup since its smaller memory budget ($\mathcal{S}_{COL}$= 100) biases optimal grouping to contain single attribute groups.



(a) Row store latencies by size *(b) Column store latencies by views*

Figure 9: Effect of All Optimizations

All Sharing Optimizations: *Summary: Applying all of our sharing optimizations leads to a speedup of up to 20X for row stores, and 8X for column stores; column stores are still faster than row stores.* Based on the optimal parameters identified from the previous experiments, we combined our optimizations to obtain the best performance. For ROW, we applied all the above optimizations with n_{agg} set to the total number of measure attributes in the table, memory threshold $\mathcal{S}_{ROW}$ = 10^4 and parallelism set to 16. For COL, we set n_{agg} and number of parallel queries similarly but did not apply the group-by optimization because of low performance gains. Figure 9 shows the latency of SEEDB on SYN when all optimizations have been applied. We see that our sharing optimizations lead to a speed up of 20X for ROW (Figures 9a) and a speedup of 10X in COL (Figures 9b). Our optimizations are most effective

[1]The different memory budgets can be explained based on the different internal parameteres and implementations of the two systems.

for datasets with large sizes and many views, particularly for ROW where reduction in table scans has large benefits.

5.4 Pruning Optimizations

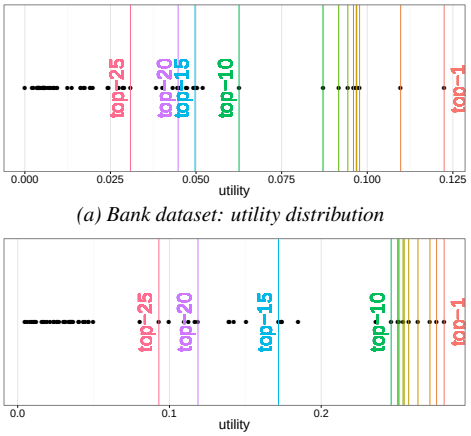
In the next set of experiments, we evaluate the impact of our pruning optimizations (Section 4.2).

Metrics: We evaluate performance of pruning optimizations along two dimensions, *latency*, as before, and result quality. We measure result quality with two metrics: (1) *accuracy*: if $\{\mathcal{V}_T\}$ is the set of aggregate views with the highest utility and $\{\mathcal{V}_S\}$ is the set of aggregate views returned by SEEDB, the accuracy of SEEDB is defined as $\frac{1}{\{|\mathcal{V}_T\}|} \times |\{\mathcal{V}_T\} \cap \{\mathcal{V}_S\}|$, i.e. the fraction of true positives in the aggregate views returned by SEEDB. (2) *utility distance*: since multiple aggregate views can have similar utility values, we use utility distance as a measure of how *far* SEEDB results are from the true top-k aggregate views. We define utility distance as the difference between the average utility of $\{\mathcal{V}_T\}$ and the average utility of $\{\mathcal{V}_S\}$, i.e., $\frac{1}{n}(\sum_i U(\mathcal{V}_{T,i}) - \sum_i U(\mathcal{V}_{S,i}))$.

Accuracy vs. Utility Distance. Since our pruning optimizations rely on utility estimates, the accuracy of pruning depends on the differences between utilities of consecutive views. Specifically, if $V_1 \ldots V_n$ is the list of aggregate views ordered by decreasing utility, then the accuracy of pruning is inversely proportional to the difference between the k-th highest utility and the $k+1$-st utility, i.e., $\Delta_k = U(V_k) - U(V_{k+1})$. Views with large Δ_k values can be pruned accurately while whose with small Δ_ks can lead to lower absolute accuracy. While this is true, notice that small Δ_ks are, in fact, the result of views at the top-k boundary having similar utility (and interestingness). For instance, the utilities at the top-5 boundary for the DIAB dataset are $U(V_5) = 0.257$, $U(V_6) = 0.254$, and $U(V_7) = 0.252$ (see Figure 10b). The small Δ_ks lead to lower accuracy for $k = 5$, but the very similar utility values indicate that V_6 and V_7 are (almost) equally interesting as V_5 . Therefore, even if V_6 or V_7 are incorrectly chosen to be in the top-k, the quality of results is essentially as high as when V_5 would have been chosen. Our *utility distance* metric correctly captures this overall quality of results. Utility distance indicates that, in the worst case, even when both V_6 or V_7 are incorrectly chosen, the overall utility of the top-5 differs only by 0.013 (≈5% error) units compared to the true top-5. As a result, we jointly consider accuracy as well as utility distance when evaluating result quality.

Techniques: In the following experiments, we evaluate four techniques for pruning low-utility views. In addition to the two pruning strategies from Section 4.2, namely the Hoeffding Confidence Intervals (CI) and the Multi-Armed Bandit (MAB), we implement two baseline strategies. The no pruning strategy processes the entire data and does not discard any views (NO_PRU). It thus provides an upperbound on latency and accuracy, and lower bound on utility distance. The other baseline strategy we evaluate is the random strategy (RANDOM) that returns a random set of k aggregate views as the result. This strategy gives a lowerbound on accuracy and upperbound on utility distance: for any technique to be useful, it must do significantly better than RANDOM. Since absolute latencies of any pruning strategy depend closely on the exact DBMS execution techniques, in this section, we report relative improvements in latency, specifically, the percent improvement in latency with pruning compared to latency without pruning. Absolute latency numbers for real datasets are discussed in Section 5.1. We do not employ early stopping for any of these techniques.

Datasets: Because pruning quality depends closely on the underlying data distribution, we evaluate our pruning optimizations on the real-world datasets from Table 1. In this section, we analyze the results for BANK and DIAB in detail; results for AIR and AIR10 are discussed in Section 5.1.

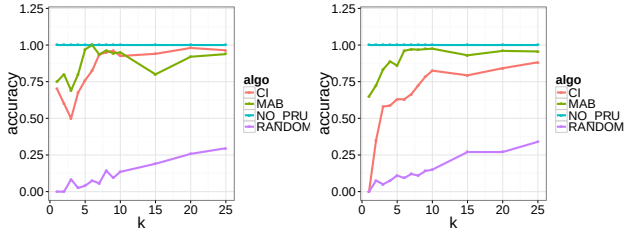


(a) Bank dataset: utility distribution

(b) Diabetes dataset: utility distribution

Figure 10: Distribution of Utilities

In each of our experiments, we vary k — the number of visualizations to recommend — between 1 and 25 (a realistic upper limit on the number of aggregate views displayed on a screen) and measure the latency, accuracy, and utility distance for each of our strategies. We pay special attention to k = 5 and 10 because empirically these k values are used most commonly. Since the accuracy and utility distance of our techniques are influenced by the ordering of data, we repeat each experiment 20 times and randomize data between runs. We report average metrics over 20 runs.



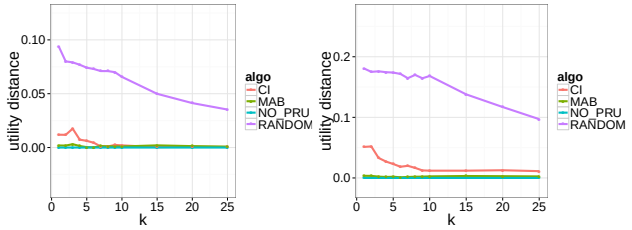
(a) Bank Dataset Accuracy *(b) Diabetes Dataset Accuracy*

Figure 11: Accuracies Across Datasets

Accuracy and Utility Distance: *Summary: The MAB and CI strategy both produce results with accuracy >75% and near-zero utility distance for a variety of k values. MAB does slightly better than CI when utlity values are closely spaced. In general, smaller Δ_k values lead to lower accuracy, but this is offset by lower utility distance that is a consequence of the smaller Δ_ks.*

BANK dataset: The distribution of utilities for all aggregate views of the bank dataset is shown in Figure 10a. In this chart, vertical lines denote the cutoffs for utilities of the top-k views where k={1,...,10,15,20,25}. The highest utility for this dataset corresponds to the *right-most* line in this chart while the 25-th highest utility corresponds to the *left-most* line. We observe that the highest and second highest utility are spread well apart from the rest (Δ_k=0.0125). The top 3rd–9th utilities are similar (Δ_k<0.002) while the 10th highest utility is well separated from neighboring utilities (Δ_{10}=0.0125). The remaining aggregate views once again have similar utilities (Δ_k<0.001). We see the effect of utility distribution in the performance of our pruning strategies. Figure 11a and Figure 12a respectively show the *average* accuracy and utility distance of our strategies over 20 runs. We find that MAB consistently produces 75% or better accuracy for all values of k and CI produces 85% or better accuracy for k>10. For k=1 and 2, the accuracy is 75% for both pruning strategies (due to large Δ_k values). The corresponding utility distance is almost zero for MAB

and about 0.015 for CI (note that these are averages). Between k=3...9, the accuracy for all strategies suffers due to small Δ_ks (< 0.002). In spite of lower accuracies, note that utility distance is consistently small (< 0.02). After k=10, the performance of all our strategies improves once again and tends to 100% accuracy and 0 utility distance. We note that NO_PRU necessarily has perfect performance, while RANDOM has extremely poor accuracy (<0.25) and utility distance (>5X that of CI and MAB).

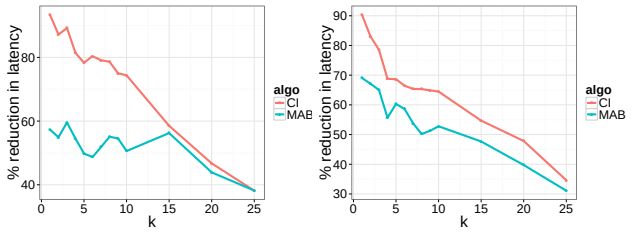


(a) Bank Dataset Utility Distance *(b) Diabetes Dataset Utility Dist.*

Figure 12: Utility Distances Across Datasets

DIAB dataset: Next, we briefly review results for the diabetes dataset. The distribution of true utilities for all aggregate views in this dataset are shown in Figure 10b. We observe that utilities for the top 10 aggregate views are very closely clustered (Δ_k <0.002) while they are sparse for larger ks. Therefore, we expect lower pruning accuracy for $k < 10$ but high accuracy for large k's. We see this behavior in Figure 11b where the accuracy of pruning is quite low ($< 60\%$) for k=1 but improves consistently to 68% (CI) and 86% (MAB) for k=5 and is >80% for $k \geq 10$. In the companion figure, Figure 12b, we see that although accuracy is relatively low k<5, utility distance is small (0.013 for CI, 0.002 for MAB) indicating that the results are high quality. Both CI and MAB produce 40X smaller utility distances compared to RANDOM.

Latency: *Summary: Both pruning strategies provide a reduction in latency of 50% or more relative to NO_PRU. For smaller k, reductions can be even higher, closer to 90%; this can be especially useful when we want to identify and quickly display the first one or two top views.* Figures 13a and 13b show the latency of our strategies for the banking and diabetes dataset. First off, we observe that the use of either of CI or MAB produces a 50% reduction in latency throughout. In fact, for CI, we obtain almost a 90% reduction in latency for small k. For k=5, MAB produces between 50 - 60% reduction while CI produces a reduction of 60 - 80%. Early stopping, i.e. returning approximate results once the top-k views have been identified, can produce even better latency reduction (results in Section 5.1). As expected, as k increases, latency also increases because we can prune fewer aggregate views.



(a) Bank Dataset Latency *(b) Diabetes Dataset Latency*

Figure 13: Latency Across Datasets

CI vs. MAB. In our evaluation, we compared two competing pruning strategies, CI and MAB. From the figures, we observe that MAB, on average, has higher accuracy and lower utility distance compared to CI, i.e., overall, it produces higher quality results. However, we find that CI performs much better than MAB on latency. Since CI can prune views more aggressively than MAB (MAB only discards one view at a time), it can rapidly prune the space of views, but this comes at the cost of result quality. Depending on the tradeoff between latency and quality of results, we can choose the best pruning strategy from CI and MAB.

6. USER STUDY

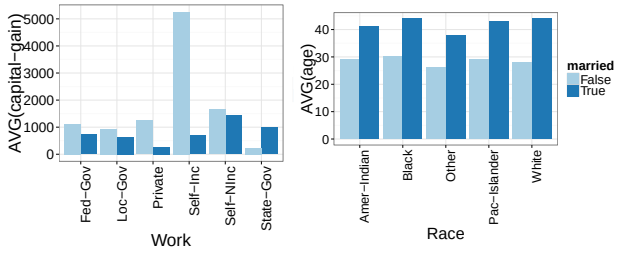
The previous section evaluated SEEDB and our optimizations in terms of performance. In this section, we assess the utility of SEEDB's recommendations with real users. First, we perform a study to validate our deviation-based utility metric. We show that although simple, our deviation-based metric can find visualizations users feel are interesting. Second, we compare SEEDB to a manual charting tool without visualization recommendations. We show that SEEDB can enable users to find interesting visualizations faster and can surface unexpected trends. We also find that users overwhelmingly prefer SEEDB over a manual charting tool.

6.1 Validating Deviation-based Utility

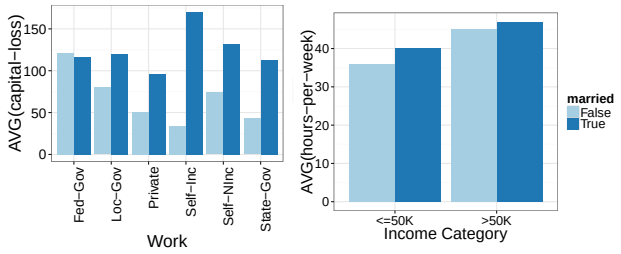
SEEDB uses deviation between the target and reference dataset as a measure of interestingness of a visualization.

Ground Truth. To validate deviation as a utility metric, we obtained ground truth data about interestingness of visualizations and evaluated SEEDB against it. To obtain ground truth, we presented 5 data analysis experts with the Census dataset (Section 1) the analysis task of studying the effect of marital status on socio-economic indicators. We presented experts with the full set of potential aggregate visualizations and asked them to classify each visualization as interesting or not interesting *in the context of the task*. Of the 48 visualizations, on average, experts classified 4.5 visualizations (sd = 2.3) as being interesting for the task. The small number indicates that of the entire set of potential visualizations, only a small fraction (~10%) show interesting trends. To obtain consensus on ground truth, we labeled any visualization chosen by a majority of participants as interesting; the rest were not. This process identified 6 interesting and 42 uninteresting visualizations. In addition to Figures 1c (interesting, recommended by SEEDB) and Figure 1d (not interesting, not recommended by SEEDB), Figure 14a, a visualization recommended by SEEDB, was labeled as interesting (according to a expert: "...it shows a big difference in earning for self-inc adults") while Figure 14b was labeled as not interesting (notice the lack of deviation). While some classifications can be explained using deviation, some cannot: Figure 14c shows high deviation and is recommended by SEEDB, but was deemed uninteresting, while Figure 14d shows small deviation but was deemed interesting ("...hours-per-week seems like a measure worth exploring").

Efficacy of Deviation-based Metric. Figure 15a shows a heatmap of the number of times a visualization was classified as interesting (*yellow* = popular, *blue* = not popular), sorted in *descending order* of our utility metric. We notice that the majority of yellow bands fall at the top of the heatmap, indicating, qualitatively, that popular visualizations have higher utility. To evaluate the accuracy of SEEDB's recommendations over the Census data, we ran SEEDB for the study task, varying k between 0 ...48, and measured the agreement between SEEDB recommendations and ground truth. As is common in data mining, we computed the "receiver operating curve" or ROC curve for SEEDB, Figure 15b, depicting the relationship between the true positive rate (TPR) on the x-axis and false positive rate (FPR) on the y-axis for different values of a parameter (k in this case). TPR is the number of interesting visualizations returned as a fraction of the total number of interesting visualizations, while FPR is the fraction of recommendations that were incorrectly returned as interesting, as a fraction of the number of non-interesting visualizations. ROC curves for highly accurate

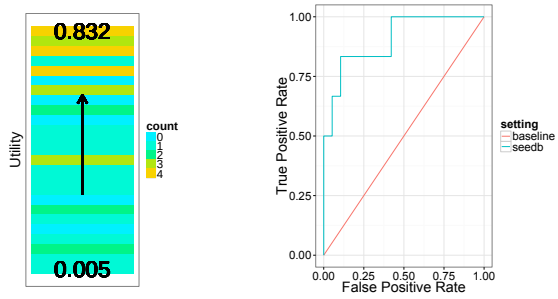


(a) High deviation, interesting *(b) Low deviation, not interesting*



(c) High deviation, not interesting *(d) Low deviation, interesting*

Figure 14: Examples of ground truth for visualizations



(a) Utility Distribution *(b) ROC of SeeDB (AUROC = 0.903)*

Figure 15: Performance of Deviation metric for Census data

classifiers are skewed towards the upper left corner of the graph. The red line indicates the random baseline (every example is classified randomly). As can be seen in the figure, SEEDB performs significantly better than the baseline. For example, for k=3, all 3 visualizations recommended by SEEDB are interesting, giving TPR = 0.5 and FPR = 0; for k=5, four of the 5 recommended visualizations are interesting, giving TPR = 4/6 = 0.667 and FPR = 0.05. The area under ROC (AUROC) for SEEDB—the typical measure of classifier quality—is 0.903. This indicates that the accuracy of SEEDB recommendations is very high.[2]

While ROC curves on different datasets and tasks will vary, this user study shows that SEEDB recommendations have high quality and coverage, despite focusing on a simple deviation-based utility metric. We expect that taking into account other aspects (apart from deviation), would improve SEEDB's recommendations even more.

6.2 SEEDB vs. Manual Visualization Tool

In this section, we describe results from a controlled user study comparing SEEDB to a manual visualization tool for performing visual analysis. We hypothesized that: (i) when using SEEDB, analysts would find interesting visualizations *faster* than when using the manual tool, (ii) analysts would find *more* interesting visualizations when using SEEDB vs. the manual tool, and (iii) analysts would *prefer* using SEEDB to a manual tool.

Participants and Datasets. We recruited 16 participants (5 female, 11 male), all graduate students with prior data analysis experience and visualization experience (e.g. R, matplotlib or Excel). None of the participants had previously worked with the study datasets.

Our study used the Housing and Movies datasets from Table 1. These datasets were chosen because they were easy to understand and comparable in size and number of potential visualizations.

Study Protocol. Our study used a 2 (visualization tool) X 2 (dataset) within-subjects design. The visualizations tools used were SEEDB and MANUAL, a manual chart construction-only version of SEEDB (i.e., SEEDB with the recommendations bar, component "D" in Figure 3, removed). Using the same underlying tool in both modes allowed us to control for tool functionality and user interface. We used a within-subjects design to compensate for per-participant differences in data analysis expertise, and used counterbalancing to remove any effects related to order and the test dataset.

Our study began with a short tutorial on the two study tools. Following the tutorial, participants were asked to perform two visual analysis tasks, one with SEEDB, and one with MANUAL. In either case, we introduced participants to the test dataset and the analytical task using written instructions. Each analytical task asked participants to use the specified tool to find visualizations supporting or disproving a specific hypothesis. Participants were asked to use the bookmark button (in component "C" in Figure 3) to flag any visualizations they deemed interesting in context of the task. Participants were also encouraged to think aloud during the study. Since the analytical tasks were open-ended, we capped each analysis session at 8 minutes. Participants filled out a tool-specific survey at the end of each task and an exit survey at the end of the study. Most survey questions were answered on a 5-point Likert scale. The study lasted ~45 minutes and participants were compensated with a $15 gift card. All studies were conducted in a lab setting using Google Chrome on a 15-inch Macbook Pro.

Methods and Metrics. Over the course of each study session, we collected data by three means: interaction logs from each tool, responses to surveys, and exit interview notes. The interaction logs capture the number of visualizations constructed, the number of visualizations bookmarked, bookmark rate, and interaction traces. SEEDB and MANUAL both support the construction of different types of charts such as bar charts, scatterplots etc. Since SEEDB can only recommend aggregate visualizations shown as bar charts, we report results for aggregate visualizations. We evaluate statistical significance of our results using ANOVA, and supplement interaction analysis with qualitative observations.

Results. Over the course of our study, participants built over 220 visualizations and bookmarked 70 visualizations (32% bookmark rate). We next describe our key findings and observations.

1. SEEDB enables fast visual analysis. Table 2 shows an overview of the bookmarking behavior for each tool focusing on total number of visualizations generated, number of bookmarks and bookmarking rate. First, we observe that the total number of (aggregate) visualizations created in the SEEDB condition is higher than that for MANUAL. While not statistically significant, this difference suggests that analysts are exposed to more *views* of the data with SEEDB than MANUAL, possibly aiding in a more thorough exploration of the data. Next, we find that the number of aggregate visualizations bookmarked in SEEDB is much higher (3X more) than that for MANUAL. In fact, the two-factor analysis of variance shows a significant effect of tool on the number of bookmarks, $F(1,1) = 18.609$, $p < 0.001$. We find no significant effect of dataset, $F(1, 1) = 4.16$. $p > 0.05$, or significant interaction between tool and dataset. While this result indicates that analysts bookmark more visualizations in SEEDB, we note that the num-

[2]AUROC's above 0.8 are considered very good, while those above 0.9 are excellent

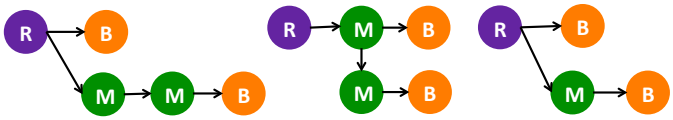


Figure 16: Interaction trace examples: (R) = Recommendation, (M) = Manual, (B) = Bookmark

ber of bookmarks for a tool may be affected by the total number of visualizations built with the tool. Therefore, to account for variance in the total number of visualizations, we also examine $bookmark\_rate$ for the two tools defined as the fraction of created visualizations that are bookmarked ($\frac{num\_bookmarks}{total\_viz}$). We find, once again, that the $bookmark\_rate$ for SEEDB (0.42) is 3X larger than the $bookmark\_rate$ for MANUAL (0.14). The two-factor analysis of variance shows a significant effect of tool on bookmark rate, F(1,1) = 10.034, p < 0.01. As before, we find no significant effect of dataset on bookmark rate, F(1, 1) = 3.125. p > 0.05, or significant interaction between tool and dataset. Together the two results above indicate that there is a **significant effect of tool on both the number of bookmarks as well as the bookmark rate**. SEEDB-recommended visualizations are 3 times more likely to be interesting compared to manually constructed visualizations. Finally, 87% of participants indicated that SEEDB recommendations sped up their visual analysis, many alluding to the ability of SEEDB to "...quickly deciding what correlations are relevant" and "[analyze]...a new dataset quickly".

	total_viz	num_bookmarks	bookmark_rate
MANUAL	6.3 ± 3.8	1.1 ± 1.45	0.14 ± 0.16
SEEDB	10.8 ± 4.41	3.5 ± 1.35	0.43 ± 0.23

Table 2: Aggregate Visualizations: Bookmarking Behavior Overview

2. All participants preferred SEEDB to MANUAL. 100% of all users preferred SEEDB to MANUAL for visual analysis, i.e., all users preferred to have recommendation support during analysis. 79% of participants found the recommendations "Helpful" or "Very Helpful" and thought that they showed interesting trends. In addition, a majority of users found SEEDB a powerful means to get an overview of interesting trends and starting points for further analysis. One participant noted that SEEDB was "...great tool for proposing a set of initial queries for a dataset". 79% of participants also indicated that SEEDB visualizations showed unexpected trends (e.g., the difference in capital gain in Figure 14a), and indicated that SEEDB suggested visualizations they wouldn't have created, e.g., although users did not manually generate Figure 1c, it was identified as an interesting visualization in the ground truth. An intriguing observation from two participants was that while they wanted recommendations to support them in analysis, they did not want to rely too heavily on recommendations. One participant noted *"The only potential downside may be that it made me lazy so I didn't bother thinking as much about what I really could study or be interested in"*. This observation suggests lines for future work that can find the right balance between automatically recommending insights and allowing the user to leverage their intuition and creativity.

3. SEEDB provides a starting point for analyses. To our knowledge, SEEDB is the first tool to provide recommendations for supporting visual analysis. As a result, we were interested in how recommendations could fit into the analytical workflow. While a participant's exact workflow was unique, we repeatedly found specific patterns in the interaction traces of SEEDB. Figure 16 shows examples of three such traces. Interaction traces show that participants often started with a recommended visualization, examined it, modified it one or more times (e.g. by changing to a different aggregate function or measure attribute) and bookmarked the resulting visualization. Thus, even if participants did not bookmark recommendations directly, they often created small variations of the visualization and bookmarked them. In other words, along with providing recommendations that were interesting by themselves, SEEDB helped direct participants to other interesting visualizations by *seeding* their analysis. This pattern was highlighted in user comments as well; e.g., "...would be incredibly useful in the initial analysis of the data", "...quickly deciding what correlations are relevant and gives a quick peek", "...great tool for proposing a set of initial queries for a dataset". In addition to understanding the role recommendations played in analysis, these observations also served to reinforce the design choice of SEEDB as a complement to manual chart construction; the mixed-initiative nature of the tool is essential for it to be functional in visual analysis.

6.3 Limitations

Given that both studies described above were conducted in the lab, the studies had limitations. First, due to constraints on time and resources, the sample sizes for both studies were small. A larger set of participants and spread of datasets could be used to further demonstrate the efficacy of our system. Second, our user studies were conducted with graduate students participants. Consequently, our results represent the perspective of capable data analysts who have limited familiarity with the data. We find that SEEDB is particularly well suited for this particular setting of initial data analysis when the user is not very familiar with the data. It would be instructive to evaluate SEEDB on datasets about which users have expert knowledge. Finally, we note that being a research prototype, the limited functionality of SEEDB and potential issues of learnability may have impacted our study.

7. UTILITY METRICS: DISCUSSION

We now discuss various aspects of our choice of utility metrics, including: *(a) Distance Functions,* i.e., using different distance functions for measuring deviation, *(b) General Utilities,* i.e., how our techniques can be used to capture a variety of other aspects of visualization recommendation beyond deviation, and *(c)* SEEDB*'s Capabilities,* i.e., how well SEEDB captures different distance functions and general utility metrics. Specifically, we compare two metrics to EMD and study the impact on: *(i)* performance and *(ii)* accuracy when pruning optimizations are employed.

7.1 Distance Functions

By default, SEEDB uses Earth Movers Distance (EMD) to compute distance between the target and reference probability distributions. However, as mentioned in Section 2, other distance functions such as Euclidean distance, K-L divergence and J-S divergence may also be used in place of EMD. Note that these are all *consistent distance functions*, like we described in Section 4.2.

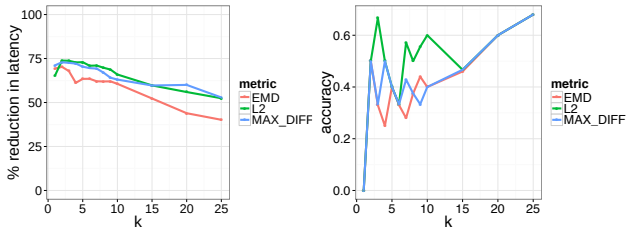
We applied each of these distance functions—Euclidean, K-L divergence, and J-S divergence—to our test datasets and compared the resulting visualization orders to that of EMD. We found a high degree of agreement on the visualization orders produced by multiple distance functions. For example, for the census dataset, when the visualization order produced by EMD was compared to that for other distance functions, we found that: (a) 3 of the top-5 visualizations were output by all three distance functions, and 4 of the top-5 were output by two of three distance functions. (b) for k=10, 6 of the top-10 visualizations were output by all three functions while 9 of the top-10 were output by two of the three distance functions. We find similar agreement on the lowest ranked visualizations as well.

Thus we find that the choice of specific function used to measure deviation is not crucial; the supported functions produce very similar visualization orders.

7.2 Pruning on Other Utility Metrics

In Section 4.2, we described the two pruning strategies used by SEEDB to rapidly discard low-utility views. As seen in Section 5, our strategies work well for the EMD-based utility metric.

We now present results which indicate that our pruning strategies are general and can support other utility metrics as well. To validate this claim, we evaluate SEEDB using two other metrics: (a) Euclidean distance (L2): like EMD, this is a deviation-based utility metric. (b) Max deviation (MAX_DIFF): this metric measures the maximum deviation between the normalized values for any group in the query and reference distribution. This metric would, for example, score the visualization in Figure 14a very highly.



(a) Latency *(b) Accuracy*

Figure 17: Latency and Accuracy for different Distance Functions

Figure 17a and Figure 17b respectively show the reduction in latency and accuracy for EMD, L2 and MAX_DIFF. We observe very similar trends with respect to reduction in latency and accuracy for all metrics. Since we have validated that pruning works well with EMD, the results indicate that pruning can also handle other metrics such as L2 and MAX_DIFF well. As mentioned in Section 4.2, we posit that pruning techniques work for any *consistent utility metric* where the estimate of the utility metric tends to the real value as the sample size increases.

7.3 General Utility Metrics

In addition to deviation, there are several other aspects or *dimensions* that need to be taken into account in order to make data-driven recommendations even more powerful. Here, we list these dimensions, and then describe how we augment SEEDB to incorporate these dimensions. A full exploration of these dimensions and their relative importance is left as future work.

The dimensions that determine the quality of a visualization recommendation include: 1. deviation from the baseline (*deviation*), 2. historical data (*history*), 3. variation in and distribution of the data itself (*data distribution*), 4. types and properties of the different attributes in the dataset (*metadata*), 5. visual qualities of the visualization (*aesthetics*), 6. past history of interactions with this user and other users (*user preferences*), and 7. meaning and relative importance of attributes in the data (*semantics*). We now describe how SEEDB can be augmented to support these dimensions within the utility metric.

First, consider an extension of the utility metric $U(V_i)$ denoted as $f_D(V_i) = w_d \times S_d + w_h \times S_h + w_l \times S_l$. Here, the first component corresponds to the utility from Section 2 so that $S_d = S(P[V_i(D_Q)], P[V_i(D_R)])$ takes into account the *deviation* between the target data and the reference (item 1 above). We can use the same distance function S to also capture utility based on historical context. For instance, let $S_h = S(P[V_i(D_Q)], P[V_i(D_C)])$ where $P[V_i(D_C)]$ refers to the typical value of the distribution $P[V_i(D_Q)]$, given historical data. For instance, when V_i refers to sales of a particular chair in Boston, $V_i(D_C)$ could be sales of that chair in Boston for the past 10 years. This measure would then allow us to identify whether the value for particular sales is deviating significantly from past patterns and is therefore interesting (item 2 above). Finally, we can also use the distance function to capture local trends. For instance, let $S_l = S(P[V_i(D_Q)], P'[V_i(D_Q)])$ where $P'[V_i(D_Q)]$ refers to the distribution P, but shifted slightly. For instance, if the sensor readings in the last five minutes differ greatly for current readings, the utility of V_i would be high. This component can capture the amount of rapid local changes that have happened within the distribution corresponding to $P[V_i(D_Q)]$ (item 3 above).

Next, we turn to the question of incorporating other recommendation dimensions into our utility metric, beyond items 1–3 above that are all distribution-based. Consider the following form of a generalized utility metric:

$$U(V_i) = f_{MA}(V_i) \times f_P(V_i) \times f_D(V_i)$$

Let $f_D(V_i)$ be the utility function measuring distribution-based utility of V_i (items 1—3). We can then augment $f_D(V_i)$ with $f_{MA}(V_i)$ and $f_P(V_i)$ capturing metadata and aesthetics (item 4 and 5), and user preferences (item 6) respectively. For instance, $f_{MA}(V_i)$ can capture best-practices about visualization and output a value accordingly. Similarly, let $f_P(V_i)$ be a function that models the users' preference towards seeing visualization V_i. This function can take into account past user history at both the individual and global levels. For instance, if the analyst typically looks at sales over time, $f_P(V_i)$ for sales over time may be high. Similarly, if sales is a popular attribute across all data analysts, $f_P(V_i)$ could be large for a V_i that depicts sales. We can also think of $f_P(V_i)$ as capturing in a limited way semantics that can be mined from previous user interactions. We note however that semantics (item 7 above) is one dimension that an automated recommendation system will not be able to capture fully.

In terms of the visualization recommendations, we find that $f_{MA}(V_i)$ and $P(V_i)$ are independent of the data distribution. Once a user poses a query, we can merely reuse previously computed values for these functions while making the recommendation. For SEEDB, these functions amount merely to constants in the utility function $U(V_i)$ that would assign weights to each view. Thus, in this form, SEEDB has been extended to incorporate other recommendation dimensions into the utility metric without any changes to the SEEDB framework.

8. RELATED WORK

SEEDB is draws on related work from multiple areas; we review papers in each of the areas, and describe how they relate to SEEDB.

Visualization Tools: The visualization research community has introduced a number of visual analytics tools such as Spotfire and Tableau [37, 3]. These tools do provide some features for automatically selecting the best visualization for a data set, but these features are restricted to a set of aesthetic rules of thumb that guide which visualization is most appropriate. Similar visual specification tools have been introduced by the database community, e.g., Fusion Tables [10]. In all these tools, the user must choose the data they want to visualize, requiring a tedious iteration through all sets of attributes. For datasets with a large number of attributes, it is often hard for the analyst to manually study every single attribute. In contrast, in SEEDB, our goal is to automatically recommend visualizations based on a generalized distance function, finding attribute sets that maximize the value of this function.

Partial Automated Selection of Visualizations. Profiler detects anomalies in data [18] and provides some visualization recommen-

dation functionality, but is restricted determining the best binning for the for the x axis: in particular, it decides which granularity is appropriate to bin on to depict the most interesting relationships between data. Since this is a much simpler problem than ours, sophisticated techniques are not necessary. VizDeck [19] depicts all possible 2-D visualizations of a dataset on a dashboard. Given that VizDeck generates all visualizations, it is meant for small datasets; additionally, [19] does not discuss techniques to speed-up the generation of these visualizations.

Scalable Visualizations. There has been some recent work on scalable visualizations that employ in-memory caching, sampling, and pre-fetching to improve the interactivity of visualization systems backed by databases (e.g., [8, 20]). Such techniques could be employed in our settings to further improve response times (although some of these techniques, such as in-memory caching, can only work with small datasets.)

Data Cube Materialization: Computations on *data cubes* [11] involve aggregating across multiple dimensions. Even when the number of attributes and number of distinct values of each attribute is relatively small, the space required to materialize then entire cube can be prohibitive, meaning that only a few (if any) dimensions can be pre-aggregated. There has been some work on identifying, given a query workload, which cubes to materialize within a certain storage budget, so as to minimize the amount of work to be performed when a query is provided [2, 12]. Hence, the optimization techniques underlying cube materialization are similar in spirit to our batching optimizations in Section 4.1, however, they focus on offline computation of views to minimize storage rather than efficient online optimization.

Browsing Data Cubes: There has been some work on using data mining techniques to aid in the exploration of data cubes [31, 34, 32, 26]. Sarawagi et al. [33, 32] explored the question of finding "interesting" cells in a cube. The interestingness of a cell is defined by how surprising its value is given the other values in the cube: [33] uses techniques based on a table analysis method while [32] uses techniques based on entropy to find interesting cells. These techniques generally identify sub-cubes of a cube that produce the most deviation amongst all sub-cells, analogous to SeeDB finding the (single) dimension attribute that shows the greatest variation in a given aggregation query. In contrast, SeeDB focuses on finding variation vis-a-vis a reference data set, recommending multiple views over a large set of possible visualizations.

In [31], Sarawagi proposes techniques to explain an increase or decrease in a specific aggregate by drilling down into that aggregate. In constrast, SEEDB seeks to find interesting differences between two datasets that have not yet been aggregated along any dimensions. Wu et al [41] tackle a similar problem in Scorpion, and differ for similar reasons.

Multi-Query Optimization: Our batching optimizations draw on related techniques from literature on shared scans [9] and multi-query optimization (e.g. [35]). Our problem is simpler however, since we don't have to wait for a batch of queries to arrive and all queries are simple aggregations. Finally, our pruning techniques allow us to stop evaluating some visualizations if we find that their utility is low, something other multi-query schemes cannot do.

Query Recommendation Systems: There is related work on recommending queries in databases (see [24]). Such systems are designed to help users pose users relevant queries over of a database, typically by consulting historical query workloads and using statistical similarity or recommender algorithms to refine user inputs. These techniques focus on recommending SQL queries instead of visualizations (and hence don't focus on visually relevant utility metrics.) We believe they could be integrated into a generalized utility metric inside SeeDB, although a full user-study comparing their effectiveness is outside the scope of this work.

9. CONCLUSIONS

Finding the right visualization given a query of interest is a laborious and time-consuming task. In this paper, we presented SEEDB, a visualization recommendation engine to help users rapidly identify interesting and useful visualizations of their data using a deviation-based metric to highlight attributes with unusual variation. Our implementation of SEEDB runs on top of a relational engine, and employs two types of optimization techniques, sharing-based, and pruning-based techniques, to obtain near-interactive performance. These techniques help us reduce latency by a factor of 100X, with the optimizations combining in a multiplicative way. Furthermore, our user study shows that SEEDB provides useful visualizations, that both help users find interesting visualizations with fewer iterations, and that users find help as an augmentation to a visualization system. In conclusion, SEEDB is an important first step in our exploration of automated visualization recommendation tools, paving the way toward automating the tedium of data analysis.

10. REFERENCES

[1] Tableau public, www.tableaupublic.com. [Online; accessed 3-March-2014].

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. VLDB '96, pages 506–521, 1996.

[3] C. Ahlberg. Spotfire: An information exploration environment. *SIGMOD Rec.*, 25(4):25–29, Dec. 1996.

[4] J.-Y. Audibert, S. Bubeck, et al. Best arm identification in multi-armed bandits. *COLT 2010-Proceedings*, 2010.

[5] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, May 2002.

[6] S. Bubeck, T. Wang, and N. Viswanathan. Multiple identifications in multi-armed bandits. In *Proceedings of the Thirtieth International Conference on Machine Learning*, JMLR '13, 2013.

[7] K. Chakrabarti et al. Approximate query processing using wavelets. In *VLDB*, pages 111–122, 2000.

[8] P. R. Doshi, E. A. Rundensteiner, and M. O. Ward. Prefetching for visual data exploration. In *DASFAA 2003*, pages 195–202. IEEE, 2003.

[9] P. M. Fernandez. Red brick warehouse: A read-mostly rdbms for open smp platforms. *SIGMOD Rec.*, 23(2):492–, May 1994.

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

[11] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, Jan. 1997.

[12] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. SIGMOD '96, pages 205–216, 1996.

[13] W. Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[14] E. Horvitz. Principles of mixed-initiative user interfaces. CHI'99, pages 159–166. ACM, 1999.

[15] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal*, 13(3):207–221, Sept. 2004.

[16] C. Jermaine et al. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4), 2008.

[17] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham. Worst-case performance bounds for simple one-dimensional packing algorithms. *SIAM Journal on Computing*, 3(4):299–325, 1974.

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

[19] A. Key, B. Howe, D. Perry, and C. Aragon. Vizdeck: Self-organizing dashboards for visual analytics. SIGMOD '12, pages 681–684, 2012.

[20] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *CoRR*, abs/1412.3040, 2014.

[21] T. Lai and H. Robbins. Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22, 1985.

[22] M. Garey et. al. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[23] J. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, Apr. 1986.

[24] P. Marcel and E. Negre. A survey of query recommendation techniques for data warehouse exploration. In *EDA*, pages 119–134, 2011.

[25] K. Morton, M. Balazinska, D. Grossman, and J. D. Mackinlay. Support the data enthusiast: Challenges for next-generation data-analysis systems. *PVLDB*, 7(6):453–456, 2014.

[26] C. Ordonez and Z. Chen. Exploration and visualization of olap cubes with statistical tests. VAKD '09, pages 46–55, 2009.

[27] A. Parameswaran, N. Polyzotis, and H. Garcia-Molina. Seedb: Visualizing database queries efficiently. *PVLDB*, 7(4), 2013.

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

[29] C. Re, N. Dalvi, and D. Suciu. Efficient top-k query evaluation on probabilistic data. In *in ICDE*, pages 886–895, 2007.

[30] U. M. L. Repository. Uci machine learning repository, 2015. [Online; accessed 29-June-2015].

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

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

[33] S. Sarawagi, R. Agrawal, and N. Megiddo. Discovery-driven exploration of olap data cubes. EDBT '98, pages 168–182, 1998.

[34] G. Sathe and S. Sarawagi. Intelligent rollups in multidimensional olap data. In *VLDB*, pages 531–540, 2001.

[35] T. K. Sellis. Multiple-query optimization. *ACM TODS*, 13(1):23–52, 1988.

[36] R. J. Serfling et al. Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics*, 2(1):39–48, 1974.

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

[38] M. Vartak, S. Madden, A. G. Parameswaran, and N. Polyzotis. SEEDB: automatically generating query visualizations. *PVLDB*, 7(13):1581–1584, 2014.

[39] J. Vermorel and M. Mohri. Multi-armed bandit algorithms and empirical evaluation. In *ECML*, pages 437–448, 2005.

[40] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *Proc. VLDB Endow.*, 6(8):553–564, June 2013.