

Constrained Recommendations for Query Visualizations

Ibrahim A. Ibrahim · Abdullah M. Albarrak · Xue Li

Received: date / Accepted: date

Abstract The improvement of data storage and data acquisition techniques has led to huge accumulated data volumes in a variety of applications. International research enterprises such as the Human Genome and the Digital Sky Survey Projects are generating massive volumes of scientific data. A major challenge with these datasets is to glean insights from them to discover patterns or to originate relationships. The analysis of these massive, typically messy and inconsistent volumes of data is indeed crucial and challenging in many application domains.

Hence, the research community has introduced a number of visualizations tools to guide and help analysts in exploring the data space to extract potentially useful information. However, when working with high-dimensional datasets, identifying visualizations that show interesting variations and trends in data is not trivial: the analyst must manually specify a large number of visualizations, explore relationships among various attributes, and examine different subsets of data before discovering visualizations that are interesting or insightful.

Though, exploring all possible visualizations involves complex challenges. It is a costly and time consuming process especially when the dimensionality is high. Furthermore, the rapid growth of databases becomes multifaceted in

I. Ibrahim
School of Information Technology and Electrical Engineering, University of Queensland,
Australia
E-mail: i.ibrahim@uq.edu.au

A. Albarrak
School of Information Technology and Electrical Engineering, University of Queensland,
Australia
E-mail: a.albarrak@uq.edu.au

Xue Li
School of Information Technology and Electrical Engineering, University of Queensland,
Australia
E-mail: xueli@itee.uq.edu.au

their channels and dimensionality thus, the transition from static analysis to real-time analytics represents a fundamental paradigm shift in the field of Big Data.

Motivated by the above challenges, we propose an efficient framework called *Realtime Scoring Engine* (**RtSEngine**) that assists analysts to limit the exploration of visualizations for a specified number of visualizations and/or certain execution time quote to recommend a set of visualizations that meet analysts' budgets. To achieve that, **RtSEngine** incorporates our proposed approaches to prioritize and score attributes that form all possible visualizations in a dataset based on their statistical properties such as selectivity, data distribution, and number of distinct values. Then, **RtSEngine** recommends the visualizations created from the top scored attributes. Moreover, we present visualizations cost-aware techniques that estimate the retrieval and computation costs of each visualization so that analysts may discard high-cost visualizations. We show and evaluate the effectiveness and efficiency of our proposed approaches, and assess the quality of visualizations and the overhead obtained by applying our techniques on both synthetic and real datasets.

Keywords Query Visualization · Aggregate Queries · Visual Analytics

1 Introduction

Data visualization is one of the most common tools for identifying trends and finding anomalies in Big Data. However, with high-dimensional datasets, identifying visualizations that effectively present interesting variations or patterns in the data is not a trivial task: analysts typically build a large number of visualizations optimizing for a range of visualization types, appealing features, and more before arriving at one that shows something valuable.

For datasets with large number of dimensions, it is extremely exhaustive for analysts to manually study all the dimensions; hence, interactive data visualization needs to be boosted with automated visualizations recommendation techniques. Interactive visualization analytics tools such as Tableau, ShowMe, and Fusion Tables [9, 23, 30] provide some features for automatically recommending the best visualization for a dataset. However, these features are restricted to a set of aesthetic rules (e.g., color, fonts, styles, ... etc) that guide which visualization is most appropriate.

Profiler [17], is another visualization tool which explores all data space to detect anomalies in data and recommends the best binning for the horizontal x axis of a visualization. It decides which granularity is appropriate to bin on to depict the most interesting relationships among data. Profiler [17] maintains a data cube in memory and uses it to support rapid user interactions. While this approach is possible when the dimensionality and cardinality are small, it cannot be used with large tables and ad-hoc queries with high dimensional data, which is the norm of scientific databases.

In the bio-medical data analysis domain, INVISQUE [34, 33] was proposed as a visual sense-making system to support information analysis for medi-

Table: GoCard											
Trip_ID	Dimension Attributes								Measure Attributes		
	Operators	Operations_date	run_ID	Route	Service_no	Direction	Boarding_stop	Alighting_stop	Trip length	Passengers no	Passengers by_route
1	Brisbane Transport	3-Jan-13	2360	P384	7394	Outbound	Leichhardt St - The Ridge (Stop 164)	West Ashgrove" Waterworks Road	9.1	1	205
2	Surfside Buslines	4-Jan-13	715	715	2228	Inbound	Town Centre Drive	Labrador Park Shopping Centre	12.8667	1	1354
3	Surfside Buslines	4-Jan-13	703	703	1239	Inbound	Turpin Road	Harbourtown Stop B	5.56667	2	1175

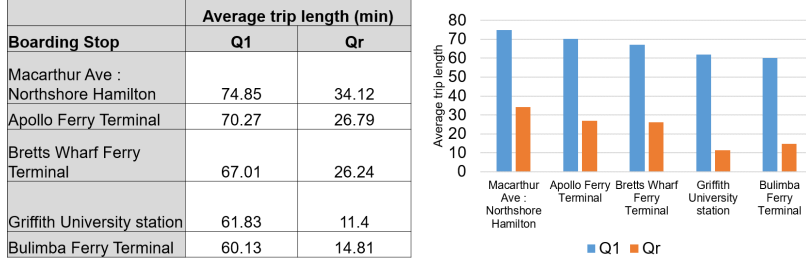
Fig. 1: Snippet from the GoCard relational database schema with a representative sample. Each row represents one trip with a bus, a ferry or a train with 12 dimensions describing the details of that trip. The database’s dimensions are classified into two: dimension attributes and measure attributes, in order to generate meaningful 2-dimensional visualizations, e.g., bar charts.

cal diagnosis. INVISQUE illustrates the similarity between the information analysis during intelligence analysis and medical diagnosis based on a Sense-Making Loop and a Data-Frame model. To overcome the challenges of exploring high-dimensional patients data, SubVIS [12] was recently proposed as a visualization tool to interactively explore bio-medical data by utilizing subspace analysis algorithms to cluster data into sub clusters and show the relationships that exist among them.

Another example of tools that recommend visualizations is VizDeck [18]. VizDeck recommends visualizations based on the statistical properties of small datasets and adopts a card game metaphor to help organize the recommended visualizations into interactive visual dashboard.

For large scale datasets, SeeDB [32] was proposed to automatically recommend interesting visualizations based on distance metrics which compute deviations among the probability distributions of the visualizations. SeeDB presents different levels of optimizations to decrease the latency and maintain the quality of visualizations such as sharing computations and combined query executions.

Although, these analytic tools present various approaches and measures to assess the interestingness of data, they still have to explore all possible visualizations to recommend a subset of interesting visualizations. Exploring the entire data space and all visualizations is almost impossible with the limited time and resources, especially when data is growing in both the dimensionality and cardinality. As a result, shifting from static analytics to realtime analytics is essential because of the rapid data accumulation when compared with a constant human cognitive capacity. Indeed this is a challenging problem. An interactive visualizations recommendation tool needs to explore the data space intelligently by discounting unnecessary visualizations and recommend only the essential ones while preserving the quality of the results.



(a) Sample results of query Q_1 and Q_r . (b) 2-D bar chart for query Q_1 and Q_r .

Fig. 2: Average trips length in minutes by boarding stop of Q_1 and the reference Q_r result into high utility value, i.e., high *interestingness*

The following example illustrates the need for an automatic visualizations technique to identify interesting visualizations from a real, large and structured database called GoCard which represents trips details of the public transportation system of the Brisbane city in Australia. Figure 1 shows a snippet of the GoCard database schema and a small sample from the database out of the 4.4 million tuples. Each tuple is a record that represents a trip using either a bus, a ferry or a train, with 12 dimensions describing that trip with more details.

Example 1 Consider a transportation analytic team that is undertaking a study for a particular alighting stop: *University of Queensland* (UQ). This stop has received a lot of passengers complaints due to poor performance, hence, it is being investigated by the team. Suppose that the team uses the GoCard database to generate 2-dimensional visualizations (e.g., bar charts) which summarize all recorded trips using different dimensions, then search for the ones that appear to explain the increase in received complaints. To accomplish that, an analyst would begin by using a program's GUI or a custom query language to execute the equivalent of the following SQL query and pull all data from the database for the alighting stop UQ:

```
Q = SELECT * FROM GoCard
WHERE alighting stop ="University of Queensland";
```

Next, the analyst would use an interactive GUI interface to generate all possible visualizations of the query result. For instance, the analyst may visualize average trip length grouped by route, total daily passengers grouped by direction, maximum trip length by boarding stop, and so on. Hence, the analyst would manually study all these visualizations to find interesting insight or visualizations that might explain the reason behind the increase of complaints. Indeed, exploring and studying all visualizations is challenging especially for high dimensional datasets. Hence, an automatic visualization recommendation technique should show the analyst the most interesting visualization based on the alighting stop UQ.

Consider the visualization for the average trip length by boarding stop: it is generated by running an operation equivalent to the following SQL query:

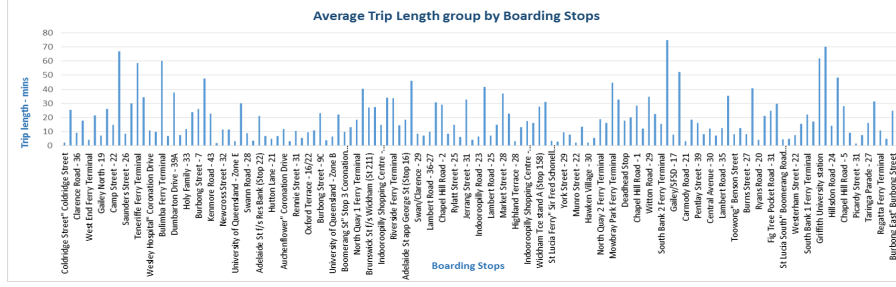


Fig. 3: 2-Dimensional bar chart visualization generated by query Q_1 . The x-axis represents the boarding stop, while the y-axis represents the average trip lengths in minutes, towards University of Queensland stop.

```

 $Q_1$  = SELECT boarding stop, AVG(trip length) FROM GoCard
WHERE alighting stop ="University of Queensland" GROUP BY
boarding stop;

```

Figure 3 shows the visualization of Q_1 's result. Consequently, the visualization in Figure 3 happened to be the most interesting visualization. The reason is, when Q_1 's result is compared with entire data, it depicts long average trip length in some boarding stops which travels towards UQ that are significantly different from the equivalent average of the trip lengths (equals 17.6 minutes) in the entire dataset. Specifically, Q_1 's result is compared against the following reference query Q_r :

```

 $Q_r$  = SELECT boarding stop, AVG(trip length) FROM GoCard
GROUP BY boarding stop;

```

The Figures 2a and 2b show a sample results of Q_1 and Q_r . \square

Example 1 above suggests that visualizations that portray trend deviations from a reference are potentially remarkable and of high interest.

Here, the average trip length grouped by boarding stops (Figure 3) is considered as the top interesting visualization, among other visualizations such as total daily passengers grouped by direction, maximum trip length grouped by boarding stop, and so on. The reason is, it depicts long average trip length in some boarding stops which travels towards UQ that are significantly different from the equivalent average of the trip lengths (equals 17.6 minutes) in the entire dataset. As listed in Figure 2a, ferry terminals scored longer trips to UQ than bus stops because ferries often take longer waiting times among stops than buses.

We summarize our contributions as follows:

- Proposing a new problem which address the limitation of current visualizations recommendation tools. Particularly, we include budget constraints to automatically recommend top- K interesting visualizations according to an input query within the specified budget.

- Designing an efficient framework called *Realtime Scoring Engine (RtSEngine)* that limits the exploration of visualizations by assessing priorities of the recommended views according to their deviation utilities and costs.
- Proposing efficient algorithms which utilize statistical features of the views such as number of distinct values, selectivity ratios, and data distribution, to early prioritize the views.
- Proposing efficient algorithms to approximate the retrieval and computations costs of the generated visualizations and evaluate their estimated costs against their deviation utilities to recommend high accuracy views in the specified budgets.
- Conducting extensive experiments that demonstrate the efficiency and effectiveness of our proposed algorithms on real and synthetic data set.

This paper is organized as follows: Section 2 describes related works on query visualization. Then, Section 3 provides preliminary details on recommendation of query visualizations and present our problem statement. Then, we present our framework *RtSEngine* in Section 4 that contains two main modules: Priority Evaluator and Cost Estimator, which recommend a set of visualizations efficiently within the specified constraints. Section ?? shows experiment results for our proposed algorithms on two real datasets.

2 Related Work

Interactive Data Visualization Tools: have interested the research community over the past few years, and it has presented a number of interactive data analytics tools such as ShowMe, Polaris, and Tableau [17, 23, 18, 6]. Similar visualization specification tools have also been introduced by the database community, including Fusion Tables [9] and the Devise [21] toolkit. Unlike SeeDB, which recommends visualizations automatically by exploring the entire views space, these tools place the onus on the analyst to specify the visualization to be generated. For datasets with a large number of attributes, it is unfeasible for the analyst to manually study all the attributes; hence, interactive visualization needs to be augmented with automated visualization techniques.

A few recent systems have attempted to automate some aspects of data analysis and visualization. Profiler is one such automated tool that allows analysts to detect anomalies in data [17]. Another related tool is VizDeck [18], in given a dataset, depicts all possible 2-D visualizations on a dashboard that the user can control by reordering or pinning visualizations. Given that VizDeck generates all visualizations, it is only meant for small datasets; and VizDeck does not discuss techniques to speed-up the generation of these visualizations.

To support visual sense-making in medical diagnosis, INVISQUE[34, 33] is an interactive visualization system proposed such as physical index cards on a two dimensional workspace. INVISQUE provides some features to support annotating, re-visiting, and merging two clusters. It discusses essential problems in designing medical diagnostic displays that can improving the review of a patients medical history [34]. A recent work, SubVIS [12] is a visualization tool

which assists the user to analyze and interactively explore computed subspaces to discover insights in highly dimensional and complex patient’s datasets. SubVIS [12] introduces an analysis workflow to visually explore subspace clusters from various perspectives and it tackles some subspace clustering challenges such as difficulty of interpretation patient results, redundancy detection in subspaces and clusters, and multiple clustering results for different parameter settings.

Statistical analysis and graphing packages such as R, SAS and Matlab could also be used generate visualizations, but they lack the ability to filter and recommend ‘interesting’ visualizations.

OLAP: there has been some work on browsing data cubes, allowing analysts to variously find *explanations* for why two cube values were different, to find which neighboring cubes have similar properties to the cube under consideration, or get suggestions on what unexplored data cubes should be looked at next [14, 26, 27].

Database Visualization Work: Fusion tables [9] allow users to create visualizations layered on top of web databases; they do not consider the problem of automatic visualization generation. Devise [10] translated user-manipulated visualizations into database queries.

Although the aforementioned approaches provide assistance in query visualization, they lack the ability to automatically recommend interesting visualizations, except SeeDB which provides different optimization techniques to automatically recommend interesting visualizations while avoiding unnecessary visualizations by utilizing two kinds of optimization techniques as explained next.

Visualizations Pruning in SeeDB: SeeDB implement an execution engine to reduce latency in assessing the collection of aggregate views which it 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. SeeDB developed a phased execution framework, each phase operates on a subset of the dataset. Phase i of n operates on the i^{th} of n equally-sized partitions of the dataset. The execution engine begins with the entire set of aggregate views as follows: During phase i , the SeeDB [32] modifies partial results for the views still under consideration using the i^{th} fraction of the dataset. The execution engine applies sharing-based optimizations to minimize scans on this i^{th} 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 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. 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:** The first pruning scheme uses worst-case statistical confidence intervals to bound views utilities. This technique is similar to top-k based pruning algorithms developed in other contexts [28]. It works as follows: during each phase, it keeps 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, it applies 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.
- **Multi-Armed Bandit Pruning:** Second pruning scheme employs a Multi-Armed Bandit strategy (MAB) [32,2]. In MAB, an online algorithm repeatedly chooses from a set of alternatives over a sequence of trials to maximize reward. This variation is identical to the problem addressed by SeeDB: the goal is find the visualizations (arms) with the highest utility (reward). Specifically, SeeDB adapts the Successive Accepts and Rejects algorithm from [2] to find arms with the highest mean reward. At the end of every phase, views that are still under consideration are ranked in order of their utility means. We then compute two differences between the utility means: Δl is the difference between the highest mean and the $k + 1^{st}$ highest mean, and Δn is the difference between the lowest mean and the k^{th} highest mean. If Δl is greater than Δn , 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 Δn 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.

However, SeeDB pruning schemes experience some limitations, as they assume fixed data distribution [32,31] for sampling to estimate the utility of views and require large samples for pruning low utility views with high guarantees. Moreover, aggregate functions MAX and MIN are not docile to sampling-based optimizations.

Offline visualizations in SeeDB: SeeDB prunes redundant views [32] : (1) For each table, it first determines the entire space of aggregate views. (2) Next, it prunes 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 , SeeDB computes the distribution for reference views on the entire dataset . (4) The resulting distributions are then clustered based on pairwise correlation. (5) From each cluster, SeeDB selects 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.

3 Preliminaries

In this section, we present background details on visualizations in the context of structural databases. We start by explaining how a visualization (or a view) is constructed by an SPJ SQL query. Then, we define our scope of visualizations that our framework is focused on, and how to measure the interestingness of a visualization based on a model proposed by [32] and another model that we believe is important. Then, we formally present our problem statement.

3.1 Background and Scope

A visualization V_i is constructed by an SQL select-project-join query with a group-by clause over a database D . The dimensions in a database table are classified into two sets: dimension attributes set $A = \{a_1, a_2, \dots\}$, and measure attributes set $M = \{m_1, m_2, \dots\}$. While the set $F = \{f_1, f_2, \dots\}$ contains all aggregation functions. Hence, each visualization V_i is represented as a triple (a, m, f) , where a is an attribute applied to the aggregation function f on a measure attribute m .

We limit our scope of visualizations on the basic components found on most 2-dimensional visualization systems such as bar charts and line charts, as they satisfy a wide range of applications requirements [16]. For instance, the Figure 2b represents a 2-D bar chart for the table in Figure 2a.

As an example, $V_i(D)$ visualizes the results of grouping the data in D by a , and then aggregating the m values using f . This view is called the *reference view*. Consequently, $V_i(DQ)$ represents a similar visualization applied to the result set denoted as DQ for a given user query Q , and is called the *target view*. An example of a target view is shown in Figure 3 where a is the boarding stops, m is the trip length, and f is the average aggregation function.

Any combination of (a, m, f) represents a view. Accordingly, we can define the total number of possible views as follows:

$$\text{View Space}(SP) = 2 \times |A| \times |M| \times |F| \quad (1)$$

Example 2 Using the GoCard database in Example 1, the dimensions within that database can be classified as follows: the set of dimension attributes is $A = \{\text{Operators, Operation date, Route, Boarding stop, Alighting stop, Direction}\}$, while the set of measure attributes is $M = \{\text{trip length, passengers by route, passengers no}\}$, and the set of aggregate functions is $F = \{\text{count, sum, avg, max, min}\}$, as shown in Figure 1. Therefore, the view space of GoCard database is: $2 \times 6 \times 3 \times 5 = 180$.

Though, in the context of Big Data, SP is potentially a very large number. Hence, there is a need to automatically score all these SP views so that exploring them become efficient and practical.

3.2 Views Utility

Each view is associated with a *utility* value. The utility of a visualization is measured as its deviation from a reference dataset D_R . For instance, visualizations that show different trends in the query dataset (i.e. DQ) compared to a reference dataset D_R are supposed to have high utility. The reference dataset D_R may be defined as the entire underlying dataset D , the complement of $DQ(D - DQ)$ or data selected by any arbitrary query $Q'(DQ')$.

Given an aggregate view V_i and a probability distribution for a target view $P(V_i(DQ))$ and a reference view $P(V_i(D_R))$, the utility of V_i is the distance between these two normalized probability distributions. The higher the distance between the two distributions, the more likely the visualization is to be interesting and therefore higher utility value. Formally:

$$U(V_i) = S(P(V_i(DQ)), P(V_i(D))) \quad (2)$$

Where S is a distance function (e.g., Euclidean distance, Earth Movers distance, etc). In addition, S can be the Pearson's correlation coefficient to capture interesting trends in visualizations .

Hence, the problem of visualizations recommendation is as follows [32]:

Definition 1 Given a user-specified query Q on a database D , a reference dataset D_R , a utility function $U()$, and a positive integer K . Find Top- K aggregate views V_1, V_2, \dots, V_K that have the highest utilities among all views while minimizing total computation time.

Now, we are in place to present our problem formulation for visualization recommendations.

3.3 Problem Formulation

Our proposed problem for visualization recommendations incorporates two limits (i.e., input parameters) to overcome the limitation of exploring all views.

Definition 2 Given a user-specified query Q on a database D , a reference dataset D_R , a utility function $U()$, a positive integer K , an execution time limit tl or a views number limit R where $K \leq R \leq SP$. Find Top- K aggregate views $V \equiv (a, m, f)$ which have maximum utilities $U(V)$ among all possible views in the specified limits R or tl while maximizing the accuracy among all Top- K views chosen from all SP views.

The limits tl and R in Def. 2 are added explicitly to overcome the limitation of exploring all views. The former is a time budget that any algorithm should not exceed, while the latter is an upper bound on the number of views to be explored. For instance, tl can be set to zero, and $R = SP$. That is, no limit on the execution time and no limit on the number of generated views.

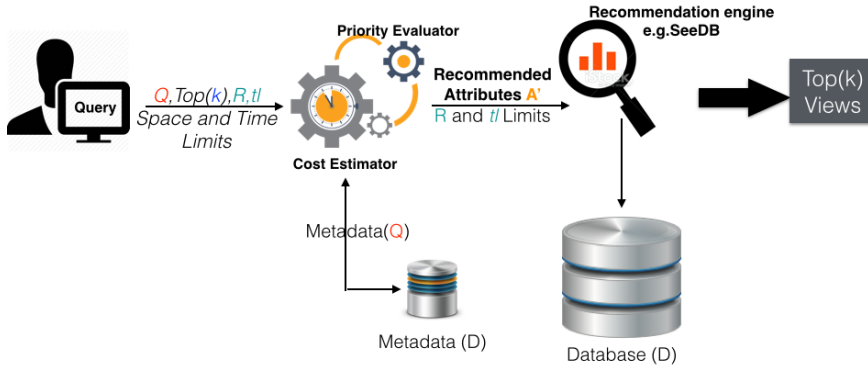


Fig. 4: *RtSEngine*: Real time Evaluation Architecture for Automatic recommendation

While those limits can be tuned by any valid value, an algorithm should output the same views as if there were no limits. This requirement makes the problem non-trivial, hence, we address it by presenting our optimization techniques encapsulated within the *RtSEngine* framework.

4 Methodology: *RtSEngine* Framework

The goal of *RtSEngine* is to recommend a set of aggregate views that are considered interesting because of their abnormal deviations. To achieve that, *RtSEngine* utilizes the following key idea: recommend views that are created from grouping high ranked dimension attributes A' within the set A . The attributes ranks in A' are computed using our proposed prioritizing techniques discussed later in the following sections. Essentially, those techniques evaluate the priorities of all dimension attributes according to their statistical features gathered from the meta-data, e.g., number of selected values, data distribution, and selectivity. Then, by reordering all dimension attributes according to their priorities, only a subset of high priority attributes are passed to the execution engine, hence, limiting the number of examined views and execution time.

Conceptually, *RtSEngine*¹, is designed as a recommendation plug-in that can be applied to any visualization engine, e.g., Tableau and Spotfire. However, in this work, we built *RtSEngine* as a standalone end-to-end system on top of SeeDB which allows users to pose arbitrary queries over data and obtain recommended visualizations. *RtSEngine* is comprised of two main modules (See Figure 4):

1. **Priority Evaluator:** An underlying module in front of any recommendation engine. Used to evaluate the dimension attributes that form visual-

¹ Implementations and data are available at:
https://github.com/ibrahimDKE/Cdb_RtsEngine_DKE_UQ

izations according to a priority function Pr computed using our proposed techniques.

2. **Cost Estimator:** This module is supposed to run in parallel with the Priority Evaluator to estimate the retrieval and computation costs of each visualization using our estimation approaches. Estimating the visualization costs in real-time improves the efficiency by discounting high costs and low priorities visualizations. Note that this module is an awareness cost approach which incorporates the estimated costs to assess visualizations based on their priorities and costs.

We define a notion of benefits $Benefit(V_i)$ of a view V_i as the gains from each view represented as the utility of view $U(V_i)$, compared with the time spent $Cost(V_i)$ to compute the view V_i . Formally:

$$Benefit(V_i) = \frac{U(V_i)}{Cost(V_i)} \quad (3)$$

Cost estimations of visualizations is discussed later on Section 4.2. Both modules (Priority Evaluator and Cost Estimator) read information by querying metadata to collect information about dimension attributes, e.g., number of distinct values and cardinality. Next, we describe the two modules in details.

4.1 Priority Evaluator: Dimension Attributes Prioritizing

In this section, we discuss the proposed approaches for prioritizing the dimension attributes in the both results set DQ and reference set (e.g. the entire dataset D) and suggest a set of visualizations that are likely to be interesting and score high deviation utilities in certain realtime limits such as maximum number of explored visualizations and execution time. The proposed approaches are based on our observations about the difference between the number of distinct values in the dimension attribute in the results set DQ and the entire dataset D affects on the deviation measures. In addition, other statistical features may also affect such as data distribution and selectivity; such features will be discussed in more detail in the next subsections. The following example illustrates this observation and describes how our strategies are agnostic for any recommendation system.

Example 3 Suppose a flights database keeps flights records which contains two dimension attributes such as destination airport name and airlines and one metric is arrival delays. Given the large size of the database (millions of records) contains 100 airports and 20 airlines companies, the analyst will study the average delays visualizations grouped by airports and airlines using a recommendation tool e.g. SeeDB and comparing these views with a reference set to glean insights about all flights departure from *Origin1*. These views can be expressed as SQL queries:

- V_1 : select AVG (arrival delays), airport from flights where origin='Origin1' group by airport;
- V_2 : select AVG (arrival delays), airlines from flights where origin='Origin1' group by airlines;

□

For instance, both visualizations V_1 and V_2 in Example 3 have the same number of distinct values: 10 destinations airports and 10 airlines operators. Eventually, aggregate views V_1 and V_2 will be compared to the corresponding reference views (i.e., the entire dataset D) according to a metric. In [32] for instance, it uses a deviation-based metric that calculates the distance between the normalized distributions between the target and reference views. In our Example 3, the average arrival delays of 10 destinations airports in view V_1 are evaluated against the average arrival delays of 100 destinations airports in the entire dataset D . Similarly, the average arrival delays of the 10 airlines operators in view V_2 are compared against the average arrival delays of the all 20 airlines operators in the entire dataset D . Thus, only 10 distinct values in view V_1 will be compared with equivalent values in the reference view, while the remaining 90 distinct values would have no equivalent values in the target view. As a result, those remaining 90 distinct values will be compared with zeros.

Furthermore, in view V_2 there are only 10 airlines operators that would be compared with zeros. This illustration arises a question about the impact of the difference in distinct values of views and their data deviations according to distance-based metrics. Formally, $Dval(V_i(DQ))$ is defined as the number of distinct values in a target view V_i . Consequently, $Dval(V_i(D))$ is the number of distinct values in the corresponding reference view V_i . In Example 3, $Dval(V_1(DQ)) = 10$ and $Dval(V_1(D)) = 100$. As mentioned previously, the deviation of each visualization is captured by a distance based metric that computes the distance between two probability distributions of views. That is, the deviation of a visualization V_i is its utility defined in Eq. 2: $U(V_i) = S(P(V_i(DQ)), P(V_i(D)))$. The distance metric $S()$ is a distance function such as Euclidean, Earth-Mover distance, ... , etc.

We discuss the influence of the difference in distinct values on computing the view utility $U(V_i)$ using Euclidean distance (although our experiments are using Earth Movers distance function as the default deviation measure). As shown in Eq. 4, L_2 -norm distance evaluates all aggregated values (points) in both views $V_i(DQ)$ and $V_i(D)$ to find the utility $U(V_i)$. Hence, V_1 's utility in Example 3 is obtained by computing the L_2 -norm distance between the average arrival delays (values) of destination airports (points) in $V_1(DQ)$ and all airports in $V_1(D)$ the entire dataset. Formally:

$$U(V_i) = \sqrt{\sum_{j=1}^n (V_i D(y_j) - V_i DQ(x_j))^2} \quad (4)$$

where $n > 0$ is the maximum number of points among $V_i D$ and $V_i DQ$. Since the compared views (i.e. target and reference view) may contain different number of distinct values, we denote n' as the number of records in $V_i(DQ)$ and n'' as the number of records in $V_i(D)$. Hence, we can rewrite the utility equation

Algorithm 1: *Diff_DVal*

Input: Attributes $A(a_1, a_2, \dots, a_n)$, Query Q , Views limit R
Output: Set \mathcal{H} : Highest priorities of dimension attributes
 $\mathcal{C} = \emptyset$ Set of all dimension attributes priorities ;
for $i = 1$ **to** n **do**
 $DvalD(a_i) \leftarrow$ number of distinct values of a_i in D ;
 $DvalDQ(a_i) \leftarrow$ number of distinct values of a_i in DQ ;
 $Pr(a_i) = |DvalD(a_i) - DvalDQ(a_i)|$;
 $\mathcal{C} \leftarrow Pr(a_i)$;
Sort \mathcal{C} ;
 $G = \lfloor \frac{R}{M \times F} \rfloor$ Calculate the required dimension number;
for $i = 1$ **to** G **do**
 $\mathcal{H} \leftarrow \mathcal{C}.get(i)$;
return \mathcal{H} ;

of view V_i as follows:

$$U(V_i) = \sqrt{\sum_{j=1}^{n'} (V_i D(y_j) - V_i DQ(x_j))^2 + \sum_{j=n'+1}^{n''} (V_i D(y_j) - 0)^2} \quad (5)$$

where $n' < n''$, and $n = n' + n''$. Because there are only n' values in the target view $V_i DQ$, then all subsequent points in the reference view $V_i D$, i.e., $n'' - n'$ values, would be compared with zeros. The higher the difference between distinct values in corresponding views forces much remaining values to be compared with zeros and increases the distance among views. In Example 3, the number of records n' of both target views $V_1(DQ)$ and $V_2(DQ)$ equals 10. However, the number of records in the reference views, i.e., n'' are $V_1(D) = 100$ and $V_2(D) = 20$. V_1 is expected to show higher distance (deviation) than V_2 when computing L_2 norm distance because 90 airports would be evaluated to zeros in V_1 but there are only 10 airlines operators with zero values in view V_2 . Since, every view is an aggregate group by query over a dimension attribute as described earlier, then, the number of records in each view equals the number of distinct values in the grouped dimension attribute.

Such observations can be utilized to early asses these (visualizations) views before executing the underlying queries to avoid computational costs (i.e., retrieval and deviation measure costs) by evaluating dimension attributes that contribute in creating visualizations. Furthermore, evaluating dimension attributes can also be done using other statistical properties such as selectivity and data distribution.

Further discussion of utilizing these features in our proposed approaches is presented in the next sections.

4.1.1 Ranking Dimension Attributes based on Distinct Values

Scoring dimensions based on difference of distinct values is the first class of prioritizing algorithms. This approach is referred to as *Diff_DVal*, and it is

Algorithm 2: *Sela*

Input: Attributes $A(a_1, a_2, \dots, a_n)$, Query Q , Views limit R
Output: Set \mathcal{H} : Highest priorities of dimension attributes
 $\mathcal{C} = \emptyset$ Set of all dimension attributes priorities ;
for $i = 1$ **to** n **do**
 $DvalD(a_i) \leftarrow$ number of distinct values of a_i in D ;
 $DvalDQ(a_i) \leftarrow$ number of distinct values of a_i in DQ ;
 $Pr(a_i) = DvalDQ(a_i) * Sel_{a_i}^{DQ} + (\frac{DvalDQ(a_i)}{DvalD(a_i)}) * Sel_{a_i}^D$;
 $\mathcal{C} \leftarrow Pr(a_i)$;
Sort \mathcal{C} ;
 $G = \lfloor \frac{R}{M \times F} \rfloor$ Calculate the required dimension number;
for $i = 1$ **to** G **do**
 $\mathcal{H} \leftarrow \mathcal{C}.get(i)$;
return \mathcal{H} ;

based on the basic observation about the number of distinct values of the dimension attributes in the results set DQ and the entire database D . The $Diff_DVal$ algorithm scores the dimension attributes according to the difference between the normalized distinct values of attributes in the result set DQ and the entire database D . Algorithm 1 inputs a query Q , a set of dimension attributes A , maximum views limit R , and/or execution time limit tl . Then $Diff_DVal$ obtains the number of distinct values for all dimension attributes in both results sets DQ and reference dataset D by posing underlying queries to select the count of distinct values. After getting the number of distinct values, $Diff_DVal$ computes the priority of each dimension attribute as the difference between each normalized values. Then, $Diff_DVal$ sorts all dimension attributes based on their priorities. Based on Eq. 1, $Diff_DVal$ computes the required number of dimension attributes G that creates the limit number of views R , then it returns the set \mathcal{H} of size G that contains a group of high priorities attributes.

In case there is an execution time limit tl , $Diff_DVal$ returns an ordered set of all dimension attributes based on their priorities, and then it passes the time limit tl to the recommendation visualization engine to limit the executions.

4.1.2 Scoring Dimension Attributes based on Selectivity

In this section, we discuss another variation of scoring the dimension attributes by capturing the data distribution in terms of query size and selectivity. Selectivity estimation is at the heart of several important database tasks. It is essential in the accurate estimation of query costs, and allows a query optimizer to characterize good query execution plans from unnecessary ones. It is also important in data reduction techniques such as in computing approximated answers to queries [1,8]. Databases have relied on selectivity estimation methods to generate fast estimates for result sizes [5,3,24,25].

The selectivity ratio [19] is defined as follows:

Definition 3 The degree to which one value can be differentiated within a wider group of similar values.

The selectivity ratio also known as the number of distinct unique values in a column divided by its cardinality [18]. Formally, the selectivity ratio of attribute a_i is:

$$Sel_{a_i}^B = \frac{\text{Number of distinct values of } a_i \text{ in } B}{\text{Cardinality of } a_i \text{ in } B} \quad (6)$$

where B is either the result set DQ or the reference dataset D , and $0 < Sel_{a_i}^B \leq 1$.

For the flight database in Example 3, both the result set DQ and the reference set D have a fixed number of records, which reveals that the selectivity ratio of the airlines column is usually low because we cannot do much filtering with just the 20 values. In contrast, the selectivity ratio of the airports column is high since it has a lot of unique values. Our proposed approach *Sela* utilizes the number of distinct values in the dimensions attributes and incorporates the query size to identify priorities of these dimensions by calculating a priority function $Pr()$ for each dimension attribute. Then *Sela* reorders the dimension attributes based on the priority.

Using selectivity ratio and the number of distinct values for assessing visualizations in D and DQ gives closer insights about the data characteristics such as the size (number of records) of aggregated views generated from group by attributes and the uniqueness degree of data in each dimension attribute. Again, in the flights database Example 3, DQ have 10 distinct airports out of 100 airports in the airports column. This means any visualization constructed by grouping airports column in result set DQ contains only 10 aggregated records. Hence, using the query size assists on quantifying how many records would be aggregated in each view that formed from grouping a dimension attribute. However, capturing the change of both number in distinct values and the number of aggregated records in each dimension attribute in result set DQ and reference set D is essential to identify visualizations that produce high deviations among all possible visualizations. Thus, we modified the priority function $Pr()$ in *Sela* to consider the number of records in each dimension attribute a_i and its selectivity ratio. Formally:

$$Pr(a_i) = DvalDQ(a_i) * Sel_{a_i}^{DQ} + \left(\frac{DvalDQ(a_i)}{DvalD(a_i)} \right) * Sel_{a_i}^D \quad (7)$$

The attribute priority $Pr(a_i)$ evaluates the number of distinct values for each dimension attribute in result set DQ multiplied by its selectivity ratio. This identifies the distinct values variations and the diversity through each dimension attribute when compared with the number of records. Furthermore, the same number of distinct values is assessed in the corresponding dimension attribute of the reference set D while considering the number of records.

In *Sela*, high priority dimension attributes are assumed to produce aggregate views (i.e., target views) that contain many groups (i.e., points) which

Algorithm 3: *DimsHisto*

Input: Attributes $A(a_1, a_2, \dots, a_n)$, Query Q , Views limit R
Output: Set \mathcal{H} : Highest priorities of dimension attributes
 $\mathcal{C} = \emptyset$ Set of all dimension attributes priorities ;
for $i = 1$ **to** n **do**
 Compute $HD(a_i)$ and $HDQ(a_i)$;
 $Pr(a_i) = d(HD(a_i), HDQ(a_i))$;
 $\mathcal{C} \leftarrow Pr(a_i)$;
Sort \mathcal{C} ;
 $G = \lfloor \frac{R}{M \times F} \rfloor$ Calculate the required dimension number;
for $i = 1$ **to** G **do**
 $\mathcal{H} \leftarrow \mathcal{C}.get(i)$;
return \mathcal{H} ;

are aggregated from records in the result set DQ . Also, the same high priority dimension attributes are assumed to produce aggregate views (reference views) by aggregating larger number of records in reference set D . This has direct effect on the aggregated values and the number of groups in both target and reference views which is expected to score high deviation utilities.

Although the *DiffDval* approach prioritizes dimension attributes (aggregate views) according to the number of distinct values, it is limited since it is incompetent to prioritize dimension attributes (aggregate views) when the number of distinct values remains stable in both result set DQ and reference set D . Moreover, *DiffDval* does not consider the data distribution within the attributes. To overcome this limitation, *Sela* utilizes the number of records and the selectivity ratios of dimension attributes in both datasets DQ and D .

The proposed algorithm *Sela* firstly computes the priority of each dimension attribute based on Eq. 7. Then, it sorts the dimension attributes based on the assigned priority to create a set \mathcal{H} of the top G dimension attributes. In case of execution time limit tl , *Sela* returns an ordered set of attributes with the highest priorities and passes time limit tl to the recommendation visualization engine to limit the executions.

4.1.3 Prioritizing Dimension Attributes based on Histograms

We proposed *Sela* and *DiffDVal* approaches to automatically recommend views with the highest deviations based on a priority for each dimension attributes in a star schema database D . Specifically, the proposed approaches relay on the number of the distinct values and the selectivity ratio of each dimension attribute in the compared datasets (i.e., DQ and D) to compute the attributes priorities.

However, the limitation of the proposed approaches is using the selectivity ratio to reflect the degree of variations of data in the dimension attributes while ignoring the distribution of data itself. In addition, it is difficult to prioritize dimensions that have the same distinct values or the same selectivity ratio.

Hence, we propose the *DimsHisto* approach which attempts to capture data distribution inside the dimensions attributes by creating frequency histograms and directly measuring the distance among corresponding histograms to evaluate these dimensions attributes. *DimsHisto* firstly generates frequency histograms for all dimensions attributes in each dataset. Then it computes the deviation in each dimension by calculating the normalized distances between each corresponding dimension attribute. For any star schema database D , a dimension attribute $a_i \in A = \{a_1, a_2, \dots, a_n\}$ can be represented as two frequency histograms: $H_{D(a_i)}$, and $H_{DQ(a_i)}$. Those two histograms are created by executing the following queries:

$H_{D(a_i)}$: Select *count*(a_i) from D group by a_i ;

$H_{DQ(a_i)}$: Select *count*(a_i) from DQ group by a_i ;

Then, after normalizing these histograms, the priority of each dimension attribute is computed as the distance between these two histograms:

$$Pr(a_i) = S(H_{D(a_i)}, H_{DQ(a_i)}) \quad (8)$$

Where $S()$ is a distance metric. Eventually, the dimension attributes are sorted according to their priorities.

A constructed histogram $H_{D(a_i)}$ is equivalent to all aggregate views created by aggregating any measure attribute (using aggregate function *Count*) and grouped by the dimension attribute a_i in the dataset D . Such a histogram assists in improving the performance of recommendation engines by avoiding the construction and computation of aggregate views along all measure attributes.

DimsHisto has to submit $2 \times |A|$ queries to compute the histograms of all dimensions and the computations of the distance metric. However, this step can be optimized to only $|A|$ by computing the histograms of all dimensions for the entire database offline. While *DimsHisto* can use any distance metric to compute the deviation among the views, we suggest to use the same metric to unify the metric of the deviations.

All proposed algorithms *DiffDVal*, *Sela* and *DimsHisto* have the same number of queries as the cost of retrieving data. While *DimsHisto* has additional cost for distance computations, it shows high accuracy for most of the aggregate functions such as Sum, Avg, and Count, because these functions are relative to the data frequencies. Though, *DimsHisto* is less descriptive to other aggregate functions such as Min and Max, as they are not amenable for sampling-based optimizations.

4.2 Cost Estimator: Visualizations Cost Estimation

The previous approaches rank dimension attributes according to their priorities and recommend visualizations while being oblivious to the retrieval and computational costs of those visualizations. However, visualizations created using different dimension and measures attributes have different retrieval and execution costs according to the query size, type of the aggregate functions,

Algorithm 4: *ViewsEstimate*

Input: Attributes $A(a_1, a_2, \dots, a_n)$, Measures $M(m_1, m_2, \dots, m_o)$, Functions $F(f_1, f_2, \dots, f_x)$

Output: S : set of views estimated costs

```

for  $i = 1$  to  $n$  do
  for  $j = 1$  to  $o$  do
    for  $p = 1$  to  $x$  do
       $C(VDQ) = EstCostDQ(a_i, m_j, f_p)$ ;
       $C(VD) = EstCostD(a_i, m_j, f_p)$ ;
       $C(d(VDQ, VD))$  Eq.10;
       $Cost = C(VDQ) + C(VD) + C(d(VDQ, VD))$ ;
       $S.add(Cost, (a_i, m_j, f_p))$ ;
return  $S$ ;

```

number of groups in each attribute and the time used to compute the deviation among all values in the corresponding visualizations.

This urges the need to only generate visualizations with high deviations and avoid the computation costs of the low-deviation ones. Besides differences in deviation utilities among different visualizations, each visualization exhibits different execution and retrieval costs. Furthermore, some visualizations may take long computations and retrieval time to only yield small deviation distances. The trade-off between gaining high utilities of the visualizations and their computations and querying costs is challenging because it involves the optimizations of finding high utilities visualizations while considering their costs.

The cost estimation step is essential to determine the cost of running and computing the deviations of a visualization to evaluate its costs against the utility obtained by measuring the deviation among visualizations. To improve the performance of recommendation applications, it is vital to discard visualizations that are expected to consume much retrieval and computation time while returning low deviation distances.

The cost estimation modules approximate CPU and I/O costs to combine them into an overall metric that is used for comparing alternative plans. The problem of choosing an appropriate technique to determine CPU and I/O costs requires considerable care. An early study [22] identified key roles for accurate cost estimation, such as the physical and statistical properties of data. Cost models take into account relevant aspects of physical design, e.g., co-location of data and index pages. However, the ability to do accurate cost estimation and propagation of statistical information on data remains one of the difficult open issues in query optimization [4].

We determine the cost of a view $Cost(V_i)$ as the sum of the following:

- Cost of running view $V_i(a, m, f)$ on dataset D .
- Cost of running view $V_i(a, m, f)$ on dataset DQ .
- Computation cost of the distance function $S(V_iDQ, V_iD)$.

Formally:

$$Cost(V_i) = C(V_iDQ) + C(V_iD) + C(S(V_iDQ, V_iD)) \quad (9)$$

As mentioned previously, the cost of running a view $V_i(a, m, f)$ on a database is affected by various factors. For instance, access paths and indices that are used to execute the view determine the proper execution plan, which reflects the view execution cost.

Running Cost of Views $C(V_iDQ)$ and $C(V_iD)$: refers to the retrieval cost of the results of both views V_iDQ and V_iD as discussed earlier.

Computation Cost of $C(S(V_iDQ, V_iD))$: is considered as the time spent on calculating the distance measure $S()$ for each value in both corresponding views.

The number of points that are compared in the corresponding views V_iDQ and V_iD is the maximum number of groups (bins) among these two views, and it is denoted as n . Alternatively, it equals the maximum number of distinct values in V_iDQ and V_iD attribute dimension.

Note that the cost of distance measures vary according to their computational complexity. For example, the Euclidean distance is faster than the Earth Mover (EMD) distance function. This is because EMD has a very high complexity $O(n^3 \log n)$ [15] while the complexity of the Euclidean distance is $O(n)$.

Since the computation cost depends on n and also depends on the computational complexity of the distance measure, we propose the following view cost equation:

$$C(S(V_iDQ, V_iD)) = O_d(n) \times d_t \quad (10)$$

Where O_d is the complexity of the distance measure, and d_t is the computation time used to compute a single point.

4.2.1 Retrieval Costs of Visualizations

In our context, the execution cost of views can be obtained using two different methods:

- **Actual Cost:** actual costs of the views are obtained by executing all queries to get their exact I/O costs, and calculating the deviation among the corresponding views.
- **DB Estimates:** reading the estimates of each view directly from the database engine (i.e., query optimizers).

However, our proposed cost method is not restricted to a certain cost estimation approach including methods based on sampling (e.g., [11, 20]), histograms (e.g., [13]), and machine learning (e.g., [7, 29]) which can be used to obtain the retrieval cost from independent estimation models.

Our proposed estimation algorithm *ViewsEstimate* is illustrated in Alg.4. *ViewsEstimate* takes dimensions, measures attributes, and the aggregated functions as input. Then it estimates I/O and computation time for each view V_i for both datasets and it returns the estimated costs of each view.

Table 1: Experiments parameters and their default values

Parameter	Range	Default Values
Top- K	1 - 70	10, 20
Views Limit R	1 - 100	70
Dimension Attributes $ A $	–	9, 10
Measure Attributes $ M $	–	3, 10
Aggregate Functions $ F $	count, sum, avg, min, max	–

The estimated I/O time for each view is obtained by reading the estimation of queries from the database query optimizer or using an independent cost estimation model. Then, *ViewsEstimate* calculates the computations costs of the distance measure between the corresponding views according to equation Eq. 10 to find the total estimated cost. Afterwards, *ViewsEstimate* adds up the computations cost and the I/O cost for V_i , then stores it into set \mathcal{S} .

Cost Estimator utilizes the set \mathcal{S} by defining a benefit of a dimension attribute $Benefit(a_i)$ as the priority of a_i divided by the maximum estimated cost of any view created using dimension attribute a_i , formally:

$$Benefit(a_i) = \frac{Pr(a_i)}{Cost(a_i)} \quad (11)$$

where $Cost(a_i)$ is the maximum estimated cost of any view created by grouping by a_i .

Finally, *DimsEstimate* ranks dimension attributes depending upon their benefits as computed by Eq. 11. As shown in Alg.4, *ViewsEstimate* inputs a set of dimensions and a visualization number limit R , then it iteratively calculates the priority and the cost of each dimension attribute to compute the benefit of each attribute. *ViewsEstimate* computes the number of dimension attributes G that create the limit R , and then outputs a set of high *Benefit* attributes of size G .

5 Experiments Setup

Before presenting our results, we describe the details of the conducted experiments including the used datasets, the proposed algorithms and the performance metrics which we use to measure the effectiveness and efficiency. Table 1 shows the parameters used throughout the experiments.

5.1 Datasets

We used the following real world datasets:

1. **Flights Database:** The Flights database contains flights delays in the year 2008. It was obtained from the U.S. Department of Transportation's

Bureau of Transportation Statistics (BTS) ². The database contains 250k tuples with a total of 20 dimensions: 10 dimension attributes and 10 measures attributes.

2. **GoCard Database:** This is the database we introduced in Example 1. It has 4.4 million tuples with a total of 13 dimensions.

5.2 Algorithms

We have implemented the following algorithms:

1. **SeeDB Baseline:** State-of-the-art algorithm [32] that processes the entire data without discarding any view. It thus provides an upper bound on latency and accuracy and lower bound on the error distance.
2. **SeeDB Rnd:** A modified version of SeeDB which returns a random set of K aggregate views as the result. This strategy gives a lower bound on accuracy and upper bound on error distance: for any technique to be useful, it must do significantly better than SeeDB Rnd.
3. **DiffDVal:** which prioritizes dimensions based on the number of distinct values in each dimension (Algorithm 1).
4. **Sela:** Our proposed algorithm (Algorithm 2).
5. **DimsHisto:** Our proposed algorithm (Algorithm ??).

Note that the Priority Evaluator module in our proposed RtSEngine utilizes DiffDVal, Sela and DimsHisto algorithms to prioritize visualizations. On the other hand, the Cost Estimator module implements the same three algorithms while utilizing the cost estimations approaches described earlier in Section 4.2.

5.3 Performance Metrics

We used two metrics for evaluating the results of our proposed approaches. One of these metrics is used by SeeDB [32] to evaluate the quality of the recommended views. To evaluate the quality and correctness of the proposed algorithms, we used the following metrics:

1. **Accuracy:** if $\{VS\}$ is the set of aggregate views with the highest utility, and $\{VT\}$ is the set of aggregate views returned by the baseline SeeDB, then the accuracy is defined as:

$$\text{Accuracy} = \frac{1}{|VT|} * \sum x, \text{ where } \begin{cases} x = 1 & \text{if } VT_i = VS_i \\ x = 0 & \text{otherwise} \end{cases}$$

i.e., accuracy is the fraction of true positions in the aggregate views returned by SeeDB.

² <http://www.transtats.bts.gov/>

2. **Distance Error:** since multiple aggregate views can have similar utility values, we use the utility distance as a measure of how far SeeDB results are from the true Top- K aggregate views. Formally, SeeDB [31] defines distance error as the difference between the average utility of $\{VT\}$ and the average utility of $\{VS\}$:

$$\text{Distance Error} = \frac{1}{K} \left(\sum_i U(VT_i) - \sum_i U(VS_i) \right)$$

All experiments were run on a PC machine with Windows 10, Intel CPU 2.8 Ghz and 8 GB of RAM memory. The RtSEngine and the algorithms were coded using the Java programming language, and datasets were loaded into a Postgres DBMS. The datasets along with the implementation are available online as a GitHub repository at: <http://>.

6 Experiments Results

Next, we present our results which demonstrates the efficiency and effectiveness of our proposed algorithms. Firstly, we test the quality of the results produced by our algorithms on the Flights database in Section 6.1. Then, we perform similar experiments on the GoCard database in Section 6.2, to show that the results are consistent. Later, we present our detailed results on the efficiency of our algorithms in Section 6.3. Finally, we show our experiments on the time limit parameter in Section 6.4 and cost estimation in Section 6.5.

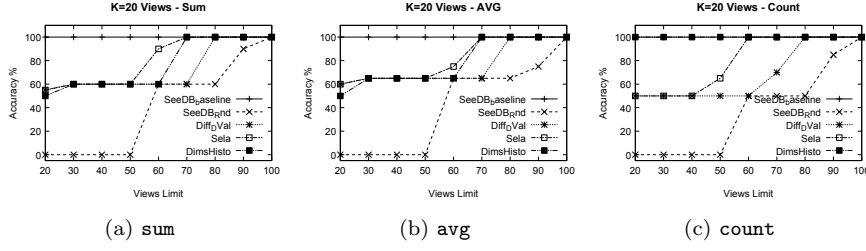
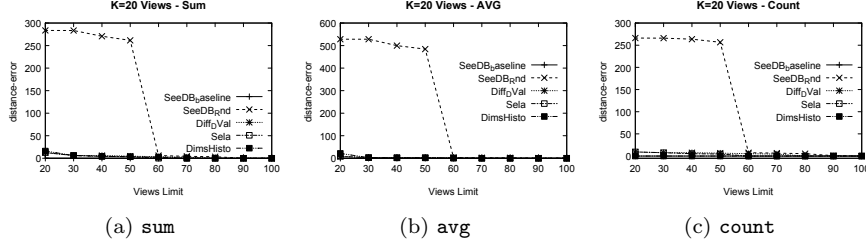
6.1 Quality Evaluation Across Aggregate Functions

In these experiments we evaluated the quality of the recommended visualizations produced by our proposed techniques across different aggregate functions, namely: `count`, `sum`, `avg`, `min`, `max`. The dataset used is the Flights database with 10 dimension attributes and 10 measures attributes. We run these experiments to assess the quality of the recommended views over each aggregate function separately with a view space size $SP = 1 \times 10 \times 10 = 100$ possible views. A utility of a view is measured using the Earth Movers Distance (EMD).

We report the accuracy and distance error of views produced by our proposed algorithms by varying the limited number of views R while $K = 20$. In these experiments, we use the following query as our target view:

```
Q: SELECT * FROM Flights WHERE uniquecarrier = 'American
    Airlines Inc.'
```

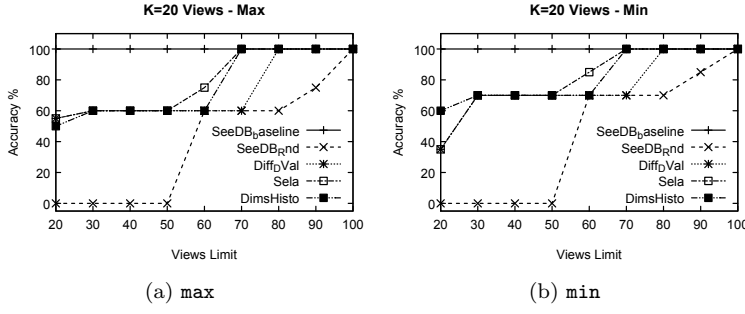
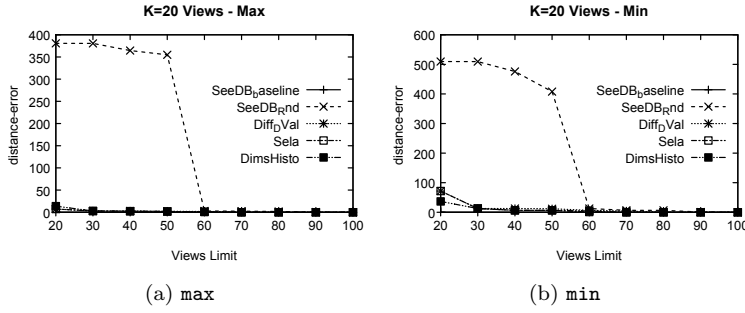
In summary, Sela and DimsHisto algorithms both produce results with accuracy $> 80\%$ for all aggregate functions, especially when $R = 60$, as shown in Figures 5a, 5b, and 5c. Moreover, they produce results with 100% accuracy

Fig. 5: Accuracy on varying view space R and $K = 20$ Fig. 6: Distance error on varying view space R and $K = 20$

when $R > 60$. *Sela* does slightly better than *DimsHisto* in terms of accuracy, as *Sela* evaluates the recommended views by capturing the change of the selectivity ratios of dimension attributes that create views in both result set and reference set. However, *DimsHisto* scores %100 accuracy in Figure 5c for aggregate function **count** because the generated histograms from this algorithm are similar to the views created by counting dimension attribute values across different measure attributes. Algorithm *Diff_Val* has the lowest accuracy and the highest distance error among the other algorithms specially for aggregate functions **max**, **min** as shown in Figures 7a and 7b as it assess recommended views based on the difference of the distinct values only.

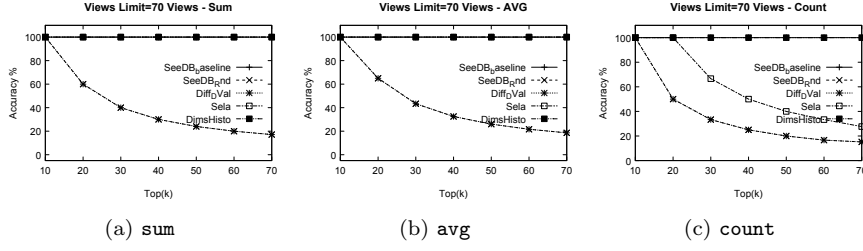
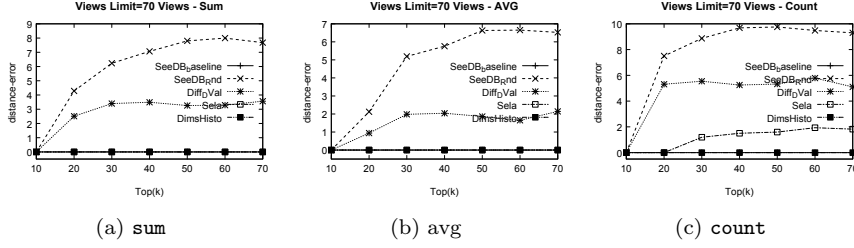
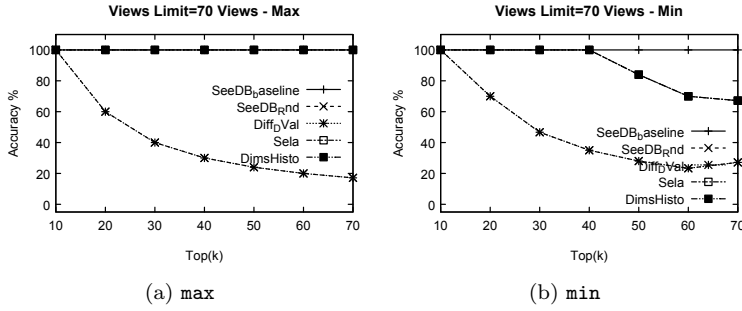
As shown in Figures 6a, 6b and 6c, the proposed algorithms produce results near-zero distance-error for all aggregate functions compared with lower baseline strategy *SeeDB_Rnd* which produce views with low quality however, the quality of the recommended views produced by the proposed algorithms is almost near to the same utilities of views output by the top baseline *SeeDBbaseline*. The distance-error of results in the first view limits=20 and 30 views as shown in Figures 8a and 8b is high specially for the aggregate function **min** because functions such as **min**, **max** are not docile for sampling but the proposed algorithms still score very low distance-error.

The proposed techniques recommend high quality views in different views limits. Furthermore, the accuracy is increasing without fluctuating along various views limits R and similarly the distance-error is declining while increasing the number of explored views R . In the worst cases, the accuracy and the

Fig. 7: Accuracy on varying view space R and $K = 20$ Fig. 8: Distance error on varying view space R and $K = 20$

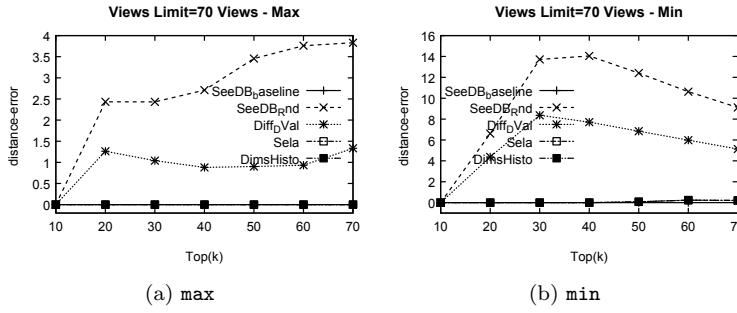
distance-error remain constant while increasing the number of explored views R .

In the following experiments, we vary K and fix the number of explored visualizations as $R = 70$ and measure the accuracy, and error-distance for each of our strategies along different aggregate functions. We pay special attention to $K = 10$ and $K = 20$ because empirically these K values are used most commonly. As the Figures 9a, 9b, 9c, 11a and 11b show, *Sela* and *Dimshisto* algorithms both produce results with accuracy %100 and zero distance-error when $K = 10$ and $K = 20$ for all aggregate functions. Moreover, *Diff_DVal* algorithm scored accuracy %100 in the first number of recommended views $K = 10$. Although, *Diff_DVal* obtains the same accuracy as *SeeDB_{Rnd}* for all aggregate functions, the *Diff_DVal* scores much better distance-error than *SeeDB_{Rnd}*, as shown in Figures 10a, 10b, 10c, 12a and 12b. As discussed in the previous experiment, the *Dimshisto* algorithm scores accuracy %100 specifically when the aggregate function is *count*. It also succeeds to recommend views with %100 accuracy and zero distance-error for aggregate functions *count*, *sum*, *avg* as shown in Figures 9a, 9b and 9c. In addition, we found that *Sela* and *Dimshisto* algorithms produce high quality views with %100 accuracy and zero distance-error for *max* aggregate function. Also, they obtain

Fig. 9: Accuracy while varying K and $R = 70$ Fig. 10: Distance error on varying K and $R = 70$ Fig. 11: Accuracy on varying K and $R = 70$

$> \%75$ and < 0.2 distance error for **min** aggregate function when $K = 70$ as shown in Figures 12a and 12b, respectively.

As Figures 10a, 10b, 10c, 12a and 12b show, the *Diff_DVal* approach scores the same accuracy produced by *SeeDB_{Rnd}* and obtains very low distance error along all aggregate functions when compared with *SeeDB_{Rnd}*. Hence, our proposed approaches boost the accuracy of the recommended views for the mostly common used K values. Moreover, the *Sela* and *DimsHisto* algorithms achieve better quality results than *Diff_DVal* because they capture the data distribution in the dimension attributes by using selectivity ratios and frequency histograms.

Fig. 12: Distance error on varying K and $R = 70$

6.2 Accuracy Evaluation

We present now our results on the GoCard database for all aggregate functions `count`, `sum`, `avg`, `min`, `max`. Hence, the view space $SP = 5 \times 9 \times 3 = 135$ views. Similar to the previous experiments, we used Earth Movers Distance (EMD) as the deviation metric for computing the utility of a view. Also, we use the following query as our target view:

Q : `SELECT * FROM GoCard WHERE alightingstop = 'University of Queensland';`

Figure 13a shows the accuracy of the results produced by algorithms *Sela*, *DiffDVal*, *DimHisto* and *SeeDB_{Rnd}* to find Top-25 views comparing with different view space R values. As shown, the proposed algorithms *Sela* and *DiffDVal* scored the same accuracy in the first 30 explored views. However, *DimHisto* shows lower accuracy than *Sela* and *DiffDVal* when the number of explored views is 45. The reason is, *DimHisto* evaluates dimension attributes according to their frequencies, hence, it is less descriptive to some aggregate functions such as `max`, `min`. Note that the accuracy of the proposed algorithms increases with R , as shown in Figure 13a. Finally, *SeeDB_{Rnd}* obtains the lowest accuracy while varying R , except when it considers almost all the views, i.e., when R approaches SP .

Figure 13b reports the distance error produced by algorithms *Sela*, *DiffDVal*, and *DimHisto* to find Top-25 ($K = 25$) views across different values of R . As shown, our proposed algorithms succeed to minimize the distance error as quickly as *SeeDB_{baseline}*, especially when expanding the space size R . Although algorithm *DimHisto* obtains lower accuracy than *Sela* and *DiffDVal* as shown in Figure 13a when $R = 45$, the distance error at the same view space is low. This is because *DimHisto* recommends different views with high utility values to minimize the distance error. *SeeDB_{Rnd}* shows high distance error even when the space size is large, i.e., $R = 90$.

To sum up, the proposed algorithms evaluate the dimension attributes according to different priorities methods. Then, by recommending a set of views which increases the quality of the view space limit R in terms of minimizing

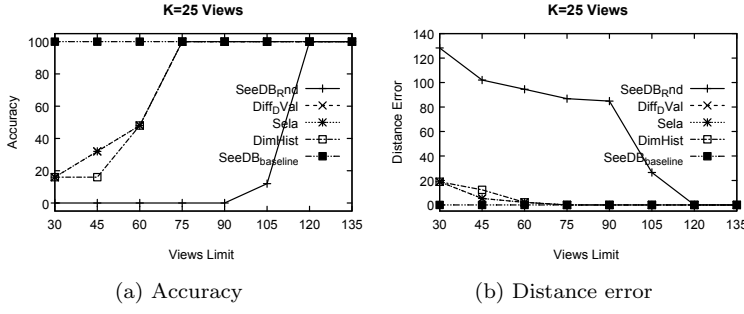


Fig. 13: Results quality while varying view space R and $K = 25$

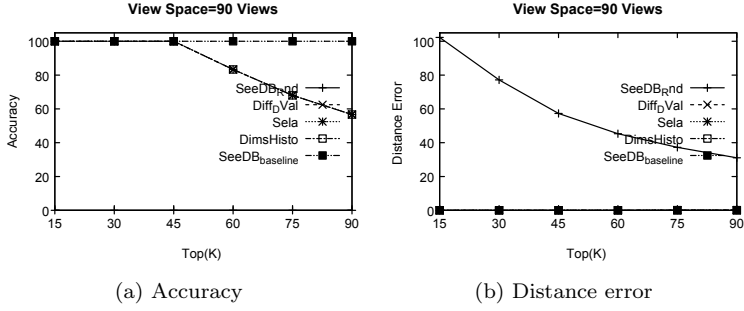


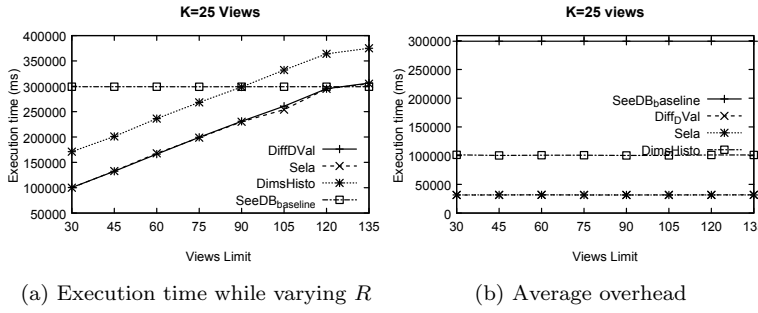
Fig. 14: Results quality while varying K and $R = 90$

the distance error and enhancing the accuracy, as explained earlier by Figures 13a and 13b.

Figure 14a shows the accuracy of the compared algorithms in a fixed space size $R = 90$ while varying K . As shown, all algorithms score %100 accuracy in the first top 45 views, which form half of the explored views. We observe that the accuracy declines while increasing K in a fixed space limit R . This is because when one dimension attribute is incorrectly prioritized, it will consequently affects all recommended views that are created from that dimension attribute. However, the accuracy is above %50 when $K = 90$ (i.e., the entire view space limit) as shown in Figure 14a. Furthermore, analysts are usually interested in recommending a small number of visualizations, i.e., $K = 25$.

In Figure 14b, the distance error of the compared algorithms are shown while varying K and R is fixed to 90 views. All algorithms produce small distance error for $K = 60$, however, *DiffDVal* shows the smallest distance error across different K values. Both *Sela* and *DimHisto* report growing distance error with respect to Top- K required by the analyst in certain view space $R = 90$.

While the discussed algorithms show high accuracy and low distance errors along different R and K values, as demonstrated above, these algorithms differentiate on the quality measures. For instance, *Sela* and *DiffDVal* obtain

Fig. 15: Algorithms performance while varying R

higher accuracy when compared with *DimHisto* as shown in Figures 13a and 14a, but *DiffDVal* obtains the lowest distance error, as shown in Figures 13b and 14b.

6.3 Efficiency Evaluation

In this section, we evaluated the efficiency of our prioritizing algorithms in terms of the overhead added to the automatic recommendation engine RtSEngine. We report the overhead as the execution time averaged over 5 runs. Similar to previous experiments, we vary K and R , and compare with the actual execution of SeeDB engine as a baseline.

As shown in Figure 15a, the total execution time of the algorithms are compared with the original SeeDB baseline. As shown, the improvements in the performance by using the proposed algorithms are significant when compared with the baseline. Furthermore, the execution time of our proposed algorithms increases linearly with R .

The execution time shown in Figure 15b is the extra overhead needed by our proposed algorithms. As shown, the average overhead is almost stable along different R values. This is because our algorithms evaluate a fixed set of dimension attributes every time, regardless of the value of R . The high cost of *DimHisto* is due to its nature: it processes a number of queries to create histograms for computing the distance among them.

The following experiments discuss the efficiency of the proposed algorithms along different K values. As shown in Figure 16a, the proposed algorithms show improvements in the execution. More than 40% when compared with the SeeDB baseline execution time. As shown above, *DimHisto* shows the highest cost among the algorithms *Sela* and *DiffDVal*.

Figure 16b shows the average overhead of the algorithms while varying K . The overhead is almost constant while increasing K . This is because the space limit R is constant too.

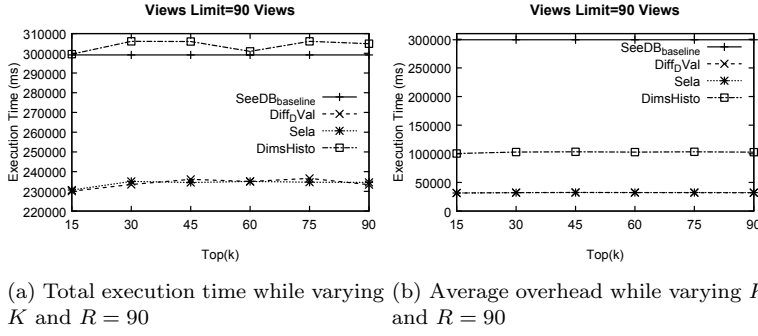


Fig. 16: Algorithms performance on varying K

6.4 Time Limit (tl)

In these experiments we evaluated the quality of the recommended visualizations produced by our proposed techniques across different aggregate functions, namely: `count`, `sum`, `avg`, `min`, `max`. The dataset used is the Flights database with 10 dimension attributes and 10 measures attributes. We run these experiments to assess the quality of the recommended views over each aggregate function separately with a view space size $SP = 5 \times 10 \times 10 = 500$ possible views. A utility of a view is measured using the Earth Movers Distance (EMD).

We report the accuracy, distance error and efficiency of views produced by our proposed algorithms by varying the time limits tl , number of views R and K . In these experiments, we use the following query:

```
Q: SELECT * FROM Flights WHERE dimmonth IN ('APR','MAY','JUN');
```

The query Q represents the second quarter of the database, so that we can compare with the entire database to find different K views while varying the time limit tl . In addition we evaluated the quality of the Top- K views produced by each algorithm with those produced by SeeDB baseline, i.e., without any time limits or optimizations used. We implemented *SeeDBTimeLimit* algorithm which processes the entire data and views in a specified execution time limit, then recommends top views that are processed in that time limit. This strategy represents a lower bound on accuracy and an upper bound on distance error.

In Figures 17a and 17b, the accuracy and distance error of the results produced by algorithms *Sela*, *DiffDVal*, *DimHisto*, and *SeeDBTimeLimit* to find a Top-100 ($K = 100$) views are compared with SeeDB baseline on different execution time limit tl . These algorithms output an ordered set of dimension attributes based on their priorities and submit the ordered set to the execution engine. Then it processes all views generated according to the ordered set that produced by algorithms. As shown, *SeeDBTimeLimit* shows high distance error and very low accuracy as well while the algorithms *Sela*

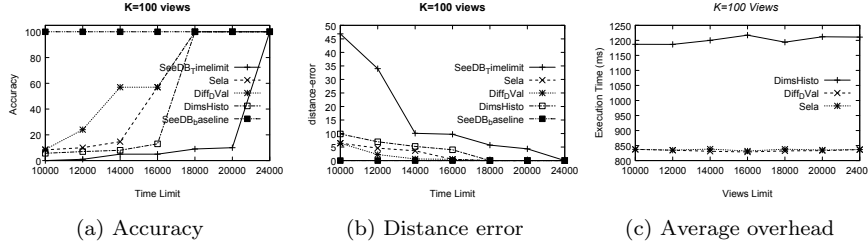


Fig. 17: Performance of *Sela*, *DiffDVal*, *DimHisto*, and *SeeDBTimelimit* on different time limits while $K = 100$

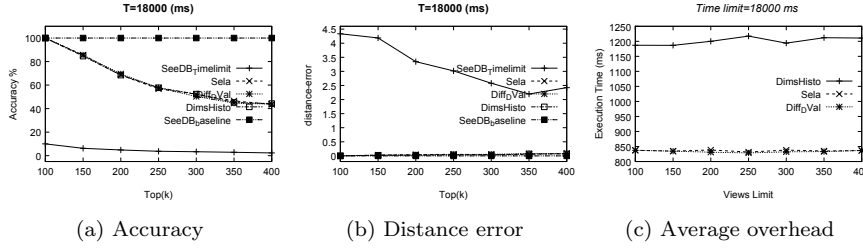


Fig. 18: Performance of *Sela*, *DiffDVal*, *DimHisto*, and *SeeDBTimelimit* on varying K and $tl = 18s$

and *DiffDVal* score higher accuracy than *DimHisto*. Although the proposed algorithms show a growing accuracy while extending the time limit, they achieved %100 accuracy for $tl > 18000$ ms. For a big database as the one used here (i.e., 500 different views), 18 seconds is considered reasonable.

The algorithms boosted the performance by more than %30 and preserved the quality of views. On the other side, Figure 17c describes the execution costs referred to as the overhead time of the proposed algorithms on the same experiment. *DimHisto* algorithm execution time is about 1200ms while the algorithms *Sela* and $N - N'$ have almost the similar execution time, about 825ms. This shows that *Sela* and *DiffDVal* algorithms are faster by %66 than *DimHisto*. As discussed previously, the additional histograms distance computations are the cause of the extra overhead in algorithm *DimHisto*.

To show the effects of varying K with time limits, Figures 18a and 18b show the accuracy of the algorithms *Sela*, $N - N'$, *DimHisto*, and *SeeDBTimelimit* in a certain time limit $tl = 18000$ ms. As shown, all algorithms score %100 accuracy in the first top 100 views. However, the accuracy declines with increasing K while tl is fixed, but the proposed algorithms score very small distance error for large values of K while *SeeDBTimelimit* shows very low accuracy and huge distance error. As illustrated in Figure 17c, the overhead costs of the proposed algorithms remain stable on different time limits. In short, the proposed algorithms improve the quality of the results, thanks to the evaluation metrics that are used along different K , R and times limits tl values. Moreover, the

algorithms overhead is comparatively small with the total execution time of baseline SeeDB.

6.5 Cost Estimation Evaluation

In the next experiments we evaluate our proposed cost estimation methods discussed previously in Section ?? on the GoCard database with $SP = 5 \times 9 \times 3 = 135$. Similar to previous experiments, we include the aggregate functions `count`, `sum`, `avg`, `min`, `max` and use the Earth Movers Distance (EMD) as our deviation metric for computing the utility. We use the following query for the next experiments:

```
Q: SELECT * FROM GoCard WHERE alightingstop ='University of
    Queensland';
```

We are interested in evaluating the results of the cost estimation methods based on the classical effectiveness and efficiency. For effectiveness, we assess the quality of views outputted by the proposed prioritizing algorithms *DiffDVal*, *Sela*, and *DimsHisto* along different cost estimation methods (i.e., DB estimate and Actual Costs) comparing with SeeDB baseline. We implemented two baseline strategies: SeeDB baseline which processes the entire data and evaluates all views without any cost considerations. Thus, it provides upper bounds on latency and accuracy and a lower bound on distance error. The other baseline strategy we implemented is Actual Costs that computes the actual execution time of all views, and also the actual computational time for computing the utility of views.

We measure the quality of results based on the accuracy and distance error. However, the efficiency of estimating methods is captured by showing the execution time across the proposed prioritizing algorithms *DiffDVal*, *Sela*, and *DimsHisto*.

The first experiment evaluates the results of the Top-25 views using the DB Estimates (reading the costs from the database optimizer) along different space limits R while comparing the estimated costs of the recommended views with the baseline. As Figure 19a shows, the accuracy of the results produced by *Sela*, *DiffDVal*, and *DimHisto* while reading the costs of the recommended views from the database optimizer to find a Top-25 views while varying R is almost %100 starting from $R = 60$.

While *Sela* algorithm has the highest accuracy and the lowest error distance among all proposed algorithms as shown in Figure 19b, the accuracy of *DiffDVal* is very low when $R \leq 60$ because it evaluates views according to the difference of distinct values only and does not consider the query size, while *Sela* does. Consequently, the error distance is higher than *Sela* and *DimsHisto*.

The following experiment illustrates the average overhead of using different cost estimations methods along our prioritizing algorithms added to the actual SeeDB baseline. In Figure 19c, the average overhead of implementing

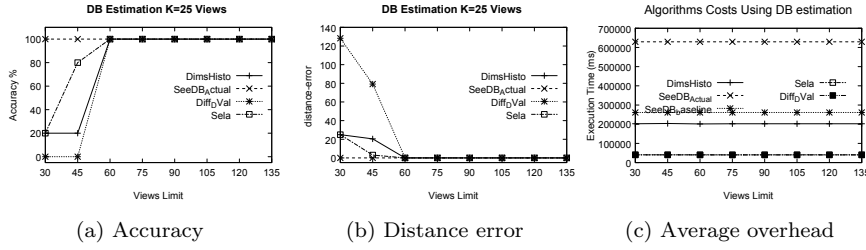


Fig. 19: Results quality and average overhead using DB Estimation

the algorithms *Sela*, *DiffDVal*, and *DimsHisto* and reading the costs from database optimizer is shown on the y-axis. As shown, computing actual costs is much expensive than running SeeDB itself. This is because SeeDB does not execute all aggregate queries. For example, the average function `avg` of a view is computed by dividing the total (`sum` aggregate function) on their frequency (`count` aggregate function). Moreover, SeeDB combines the aggregate queries of the datasets D and DQ . All algorithms have a stable performance on different space limits R because the algorithms evaluate the same set of dimension attributes A and outputs a subset A' of top scored dimension attributes. As shown, *DimsHisto* shows a considerable time cost since it create and assess histograms, however, both algorithms *Sela* and *DiffDVal* have nearly equal execution costs.

7 Conclusion

Finding top interesting visualizations by exploring a specified number of visualizations or an execution time budget, while persevering the quality and the accuracy of the recommended views is a challenging and emerging problem. In this paper, we addressed this problem and proposed an efficient framework called Realtime Scoring Engine (RtSEngine) that assist data analysts in the exploration of visualizations generated from structured databases.

Specifically, RtSEngine supports analysts by efficiently recommending visualizations while meeting analysts budgets: certain number of visualizations or execution time quote. RtSEngine accomplishes this by incorporating inventive approaches to prioritize and score attributes that form all possible visualizations in database based on their statistical proprieties such as selectivity ratio, data distribution, and number of distinct values. Then, RtSEngine recommends the views created from top scored attributes.

In addition, we presented visualizations cost-aware techniques that estimate the retrieval and computation costs of all visualizations. Those estimated costs are then fed to RtSEngine to recommend views while considering their costs to guarantee the efficiency and effectiveness of the recommendation process.

Finally, we conducted comparative experiments and demonstrated the quality of visualizations and the overhead obtained by applying our techniques on both synthetic and real datasets. The experiments showed superior effectiveness and efficiency of our proposed approaches on different time and space limits.

References

1. Barbará, D., DuMouchel, W., Faloutsos, C., Haas, P.J., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H.V., Johnson, T., Ng, R.T., Poosala, V., Ross, K.A., Sevcik, K.C.: The new jersey data reduction report. *IEEE Data Eng. Bull.* **20**(4), 3–45 (1997)
2. Bubeck, S., Wang, T., Viswanathan, N.: Multiple identifications in multi-armed bandits. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16–21 June 2013*, pp. 258–265 (2013). URL <http://jmlr.org/proceedings/papers/v28/bubeck13.html>
3. Charikar, M., Chaudhuri, S., Motwani, R., Narasayya, V.R.: Towards estimation error guarantees for distinct values. In: *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, May 15–17, 2000, Dallas, Texas, USA, pp. 268–279 (2000)
4. Chaudhuri, S.: An overview of query optimization in relational systems. In: *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS '98*, pp. 34–43. ACM, New York, NY, USA (1998). DOI 10.1145/275487.275492. URL <http://doi.acm.org/10.1145/275487.275492>
5. Chaudhuri, S., Motwani, R., Narasayya, V.R.: Random sampling for histogram construction: How much is enough? In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, June 2–4, 1998, Seattle, Washington, USA., pp. 436–447 (1998)
6. Fisher, D.: Hotmap: Looking at geographic attention. *IEEE Trans. Vis. Comput. Graph.* **13**(6), 1184–1191 (2007)
7. Getoor, L., Taskar, B., Koller, D.: Selectivity estimation using probabilistic models. In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD '01*, pp. 461–472. ACM, New York, NY, USA (2001). DOI 10.1145/375663.375727. URL <http://doi.acm.org/10.1145/375663.375727>
8. Gilbert, A.C., Kotidis, Y., Muthukrishnan, S., Strauss, M.: Optimal and approximate computation of summary statistics for range aggregates. In: *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 21–23, 2001, Santa Barbara, California, USA (2001)
9. Gonzalez, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R., Shen, W., Goldberg-Kidon, J.: Google fusion tables: web-centered data management and collaboration. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, USA, June 6–10, 2010*, pp. 1061–1066 (2010)
10. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online aggregation. In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data*, May 13–15, 1997, Tucson, Arizona, USA., pp. 171–182 (1997)
11. Hou, W.C., Ozsoyoglu, G.: Statistical estimators for aggregate relational algebra queries. *ACM Trans. Database Syst.* **16**(4), 600–654 (1991). DOI 10.1145/115302.115300. URL <http://doi.acm.org/10.1145/115302.115300>
12. Hund, M., Böhm, D., Sturm, W., Sedlmair, M., Schreck, T., Ullrich, T., Keim, D.A., Majnaric, L., Holzinger, A.: Visual analytics for concept exploration in subspaces of patient groups. *Brain Informatics* pp. 1–15 (2016)
13. Ioannidis, Y.: The history of histograms (abridged). In: *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29, VLDB '03*, pp. 19–30. VLDB Endowment (2003). URL <http://dl.acm.org/citation.cfm?id=1315451.1315455>
14. Jagadish, H.V.: Review - explaining differences in multidimensional aggregates. *ACM SIGMOD Digital Review* **1** (1999)

15. Jang, M.H., Kim, S.W., Faloutsos, C., Park, S.: A linear-time approximation of the earth mover's distance. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM '11, pp. 505–514. ACM, New York, NY, USA (2011). DOI 10.1145/2063576.2063652. URL <http://doi.acm.org/10.1145/2063576.2063652>
16. Jugel, U., Jerzak, Z., Hackenbroich, G., Markl, V.: Vdda: Automatic visualization-driven data aggregation in relational databases. *The VLDB Journal* pp. 53–77
17. Kandel, S., Parikh, R., Paepcke, A., Hellerstein, J.M., Heer, J.: Profiler: Integrated statistical analysis and visualization for data quality assessment. In: Proceedings of the International Working Conference on Advanced Visual Interfaces, pp. 547–554. ACM (2012)
18. Key, A., Howe, B., Perry, D., Aragon, C.R.: Vizdeck: self-organizing dashboards for visual analytics. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20–24, 2012, pp. 681–684 (2012)
19. Lahdenmaki, T., Leach, M.: Relational Database Index Design and the Optimizers. John Wiley & Sons (2005)
20. Lipton, R.J., Naughton, J.F., Schneider, D.A.: Practical selectivity estimation through adaptive sampling. In: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, SIGMOD '90, pp. 1–11. ACM, New York, NY, USA (1990). DOI 10.1145/93597.93611. URL <http://doi.acm.org/10.1145/93597.93611>
21. Livny, M., Ramakrishnan, R., Beyer, K.S., Chen, G., Donjerkovic, D., Lawande, S., Myllymaki, J., Wenger, R.K.: Devise: Integrated querying and visualization of large datasets. In: SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13–15, 1997, Tucson, Arizona, USA., pp. 301–312 (1997)
22. Mackert, L.F., Lohman, G.M.: R* optimizer validation and performance evaluation for local queries. *SIGMOD Rec.* **15**(2), 84–95 (1986). DOI 10.1145/16856.16863. URL <http://doi.acm.org/10.1145/16856.16863>
23. Mackinlay, J.D., Hanrahan, P., Stolte, C.: Show me: Automatic presentation for visual analysis. *IEEE Trans. Vis. Comput. Graph.* **13**(6), 1137–1144 (2007)
24. Mannino, M.V., Chu, P., Sager, T.: Statistical profile estimation in database systems. *ACM Comput. Surv.* **20**(3), 191–221 (1988)
25. Piatetsky-Shapiro, G., Connell, C.: Accurate estimation of the number of tuples satisfying a condition. In: SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18–21, 1984, pp. 256–276 (1984)
26. Sarawagi, S.: User-adaptive exploration of multidimensional data. In: VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10–14, 2000, Cairo, Egypt, pp. 307–316 (2000)
27. Sathe, G., Sarawagi, S.: Intelligent rollups in multidimensional OLAP data. In: VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11–14, 2001, Roma, Italy, pp. 531–540 (2001)
28. Serfling, R.J.: Probability inequalities for the sum in sampling without replacement. *The Annals of Statistics* pp. 39–48 (1974)
29. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: Leo - db2's learning optimizer. In: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, pp. 19–28. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2001). URL <http://dl.acm.org/citation.cfm?id=645927.672349>
30. Stolte, C., Hanrahan, P.: Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. In: Proceedings of the IEEE Symposium on Information Visualization 2000, INFOVIS '00, pp. 5–. IEEE Computer Society, Washington, DC, USA (2000). URL <http://dl.acm.org/citation.cfm?id=857190.857686>
31. Vartak, M., Madden, S., Parameswaran, A., Polyzotis, N.: Seedb: towards automatic query result visualizations. Tech. rep., Technical Report, data-people. cs. illinois.edu/seedb-tr. pdf
32. Vartak, M., Madden, S., Parameswaran, A.G., Polyzotis, N.: SEEDB: automatically generating query visualizations. *PVLDB* **7**(13), 1581–1584 (2014)
33. Wong, B.L.W., Chen, R., Kodagoda, N., Rooney, C., Xu, K.: INVISQUE: intuitive information exploration through interactive visualization. In: Proceedings of the International Conference on Human Factors in Computing Systems, CHI 2011, Extended

-
- Abstracts Volume, Vancouver, BC, Canada, May 7-12, 2011, pp. 311–316 (2011). DOI 10.1145/1979742.1979720. URL <http://doi.acm.org/10.1145/1979742.1979720>
34. Wong, B.L.W., Xu, K., Holzinger, A.: Interactive Visualization for Information Analysis in Medical Diagnosis, pp. 109–120 (2011)