SEEDB: Efficient Data Driven Visualization Recommendations to Support Visual Analytics

Chirag Uday Kamath, Divya Maiya, Neha Prakash
Github: https://github.com/Divya-Maiya/SeeDb

Problem Statement

Data visualization, the process of putting data into a chart, graph, or other visual format helps inform analysis and interpretation and also presents the analyzed data in ways that are accessible to and engage different stakeholders. However, very often, when working with high-dimensional data, it is hard to produce visualizations that consist of the most relevant aspects of the data. More specifically, the paper highlights two hindrances when it comes to data visualization systems, scale, and utility. Scale concerns analyzing a huge number of possible visualizations while responding in real-time and utility involves determining a suitable metric for evaluating how interesting visualization is.

In order to represent the problem statement in a more concrete formal manner, the paper establishes some definitions of all the parameters involved. The problem focuses on a database D with a snowflake schema. We use A (dimension attributes) to denote the attributes that would be used to group by in the visualizations. We use M (measure attributes) to denote the attributes that will be used to aggregate in the visualization. Next, we denote the set of potential aggregate functions (e.g. COUNT, SUM, AVG) over the measure attributes. Eventually, the output (visualization data) is a two-column table that can be easily visualized using standard visualization mechanisms like bar charts or trend lines.

Using the above-established definitions, we represent a visualization V_i as a function represented by a triple (a, m, f) where $a \in A$, $m \in M$, and $f \in F$. In the implementation, the utility of the visualizations is determined using deviations. In particular, the visualizations that show different trends in the query dataset compared to a reference dataset (called D_R) are said to have high utility or in other words, are interesting visualizations. Finally, the implementation supports several functions to compute the Utility U like KL divergence, Earth Mover's Distance, Euclidean Distance, and JS Distance.

We can now formally state the problem as follows[1]:
Given a user-specified query Q on a database D, a reference dataset D_R, a utility function U as defined above, and a positive integer k, find k aggregate views $V \equiv (a, m, f)$ that have the largest values of U(V) among all the views (a, m, f) while minimizing total computation time.

Techniques Implemented

Overview

As explained in the problem statement, every visualization we consider is represented by a triple (a, m, f) where $a \in A$, $m \in M$ and $f \in F$. In our implementation, we create a complex yet efficient data structure that holds all possible

[1] The formal problem statement has been referenced directly from the paper to avoid ambiguity

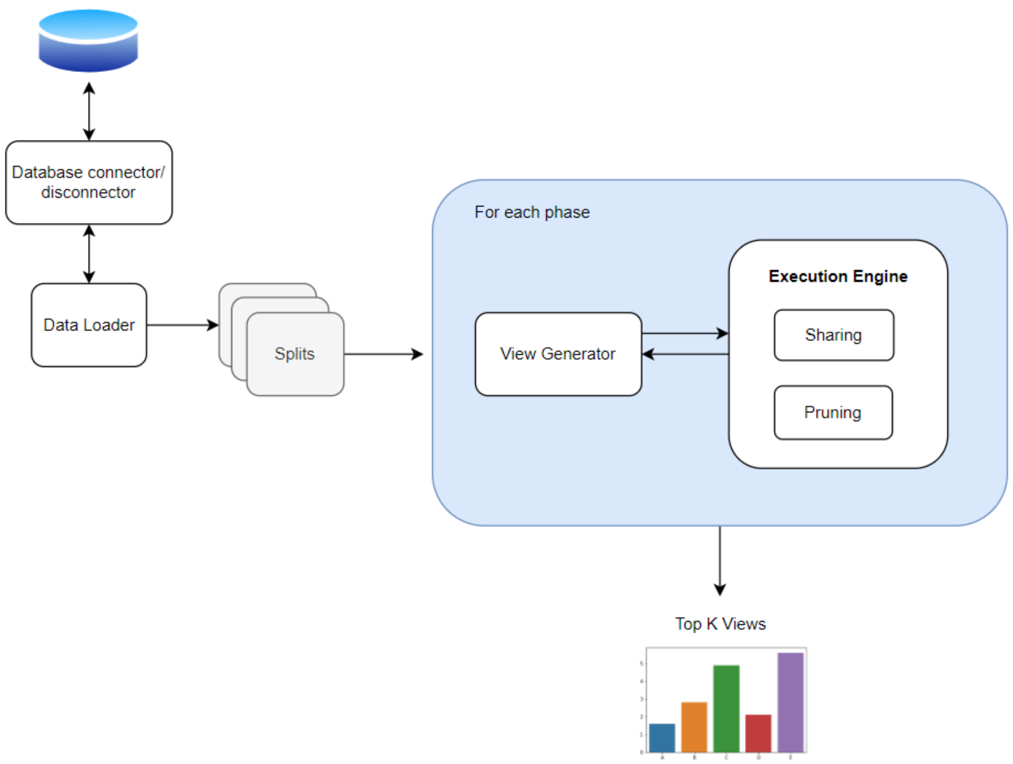
combinations of the (a, m, f) triple. We make use of this data structure to implement both sharing and pruning-based optimizations and finally for visualizing the top k views.

In our implementation, we set the values of all parameters from the problem statement as follows:

Parameter	**Value**
Dataset	Census
Target Dataset	All married groups
Reference Dataset	All unmarried groups
Dimension Attributes	workclass, education, occupation, relationship, race, sex, native_country, salary
Measure Attribute	age, capital_gain, capital_loss, hours_per_week
Aggregation Attribute	sum, min, max, avg, count
Number of splits	5
k	5

System Architecture

The following diagram depicts the system architecture of our implementation:



We make use of the following process to generate the top k interesting views:

Step 1 - Connect to the database
We use the psycopg2 module to establish a connection to the Postgres database. The entire connection process is config controlled and can be used to fit any local Postgres database by updating the database connection credentials in the *config/seedb_configs.ini* file.

Step 2 - Data loading and preprocessing
In this step we load the data from the Postgres database and perform some basic data cleansing and preprocessing. This is discussed in detail in the next section. After the data is cleaned we load the cleaned data from adults.data file into a pandas dataframe. Since SeeDB requires a random sampling of data, we shuffle the rows and then proceed to partition the dataset into *p* phases. Like most hyperparameters, the number of splits/phases is controlled by the config file and can be altered to increase/decrease the number of phases that the phased execution engine operates on. After splitting our shuffled dataset into phases, we store them temporarily in csv files and then create a new table for each of them, thus finally dividing the data into 5 partitions and passing it to the execution engine.

Step 3 - Phased execution
This is the key step of the implementation, wherein we perform a phase-wise execution to iteratively process the partitioned data. In each phase, we deal with one split of the data. We use the sharing-based optimization to fetch the required data for the target and reference queries, calculate the utility measure (like KL divergence) and then prune out the most uninteresting views using the pruning-based optimization. We discuss these optimizations in further detail in the following sections. After n iterations, we finally output the top k interesting aggregate views.

Step 4 - Visualization
Finally, we visualize the top k interesting aggregate views using the matplotlib module.

Data Preprocessing

Data preprocessing is a step performed so that raw data can be converted to a form that can be used for analysis. Data preprocessing for the Census dataset included discarding rows that had meaningless data such as ‘?’. The Census dataset is a relatively clean dataset but requires a few cleaning operations so that we get more meaningful results from SeeDB.
The following preprocessing operations were performed:

1. Since we wanted to use both adult.data and adult.test, we needed to clean adult.test as there was an extra ‘.’ at the end of each row therefore creating new values of *economic_indicator* attribute which did not exist in the actual dataset. These extra characters were removed.
2. adult.data and adult.test were merged into a single dataset
3. Empty values were replaced with the ‘?’ value as our query ignores such rows with missing values.

Utility Measure

For our base implementation, we use KL Divergence as our utility measure, to help determine the interestingness of an aggregate view. In mathematical statistics, the Kullback–Leibler divergence (also called relative entropy), is a measure of how one probability distribution Q (in our case the target set) is different from a second, reference probability distribution P (in our case the reference set). As a part of one of our extensions, we also implemented other utility measures including Earth Mover’s distance, Euclidean Distance and Jenson-Shannon Distance. We discuss this further in the extensions section.

Sharing Based Optimization

We implement the sharing-based optimization using the **Combine Multiple Aggregates** technique.
This technique takes advantage of the fact that each view is represented by a tuple (a, m, f) where a group-by is performed with attribute a to measure the attribute m by leveraging the aggregation function f. In our implementation, we combine the queries required for different combinations of (m, f) for a particular attribute a, thereby vastly reducing the number of queries fired.

Further, one of our extensions is to implement query rewriting. Since the target and reference query are essentially the same and only operate on different subsets of the original dataset, we rewrite both queries as a single query.

Pruning Based Optimization

Out of all the visualizations that could potentially be generated after each phase, most tend to be low in utility, and generating them would unnecessarily consume computational resources. Hence, at the end of each phase, we estimate the utility value of each view based on the data processed and prune out the views with low deviation which are considered low utility. We use confidence interval-based pruning, wherein the worst case confidence interval is derived from Hoeffding-Serfling inequality to calculate the low utility views. We used hyperparameter tuning to find that the best value of delta (for calculating confidence intervals) lies at 1e-5.

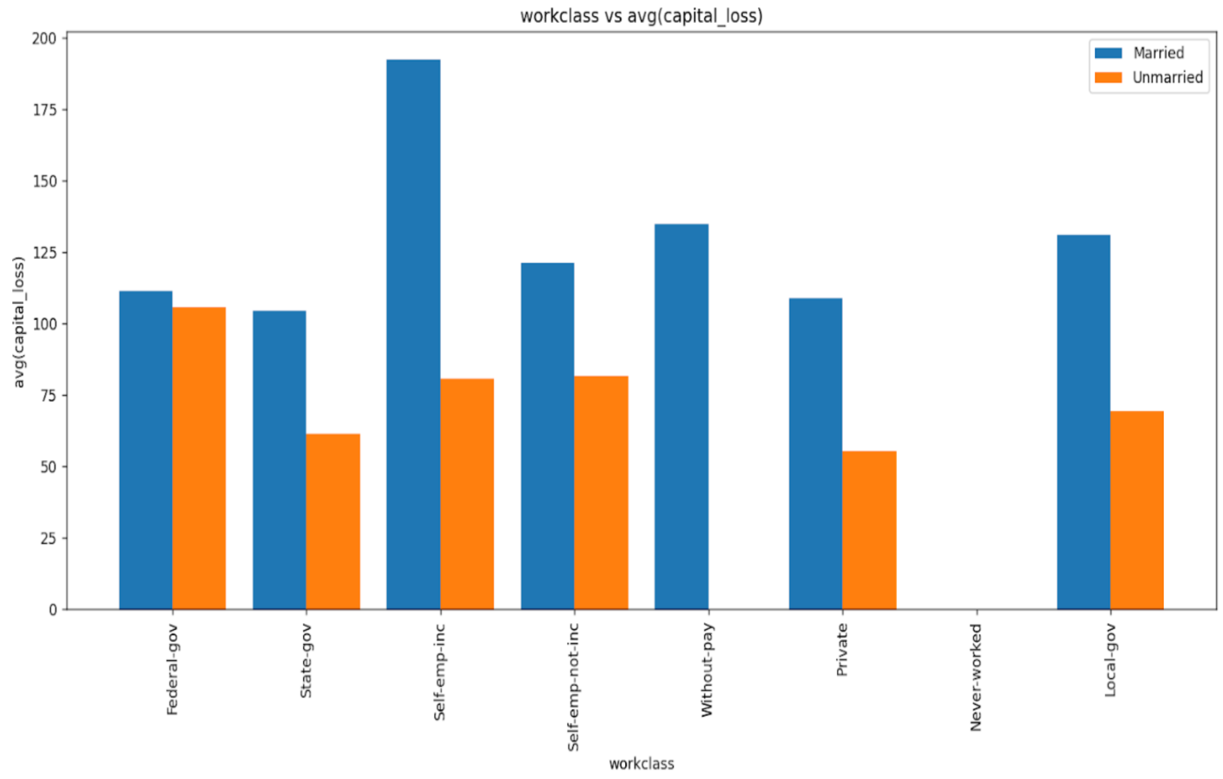
Tech Stack

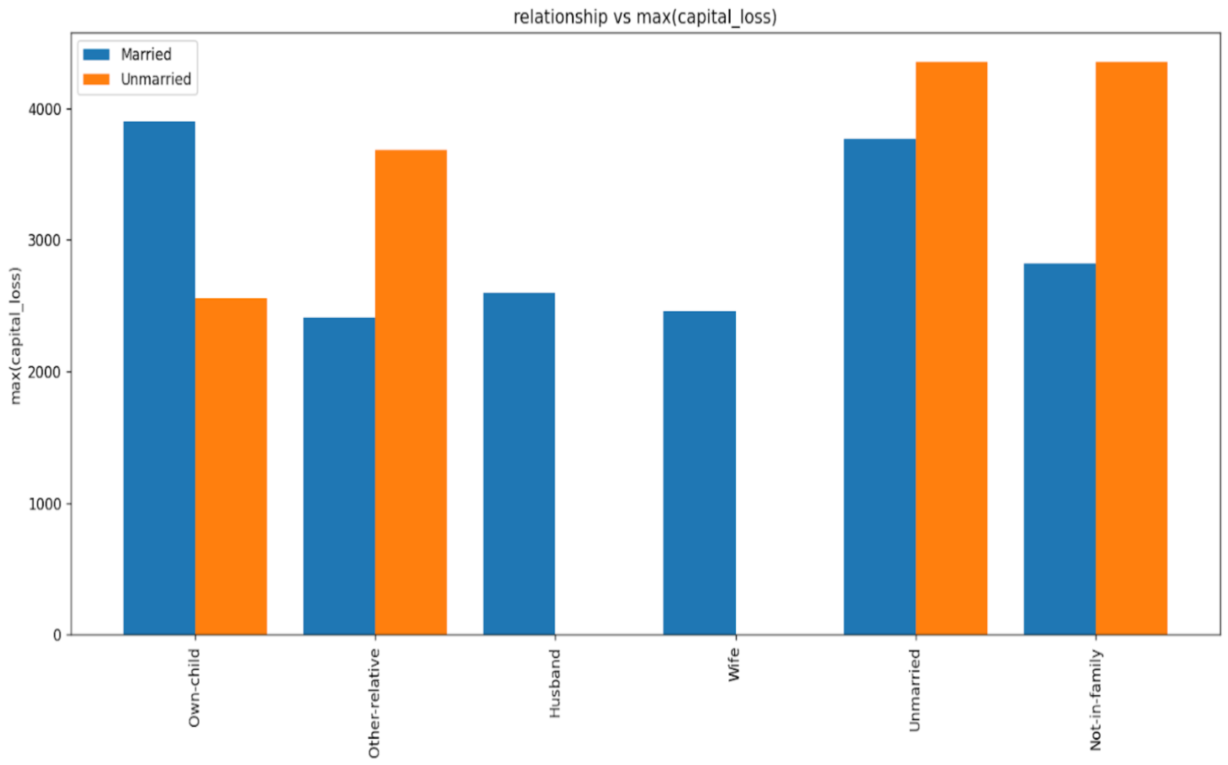
The entire project has been implemented in Python, predominantly making use of the psycopg2 library, which is one of the most popular PostgreSQL database adapters for Python. We use NumPy and pandas for processing the data and making meaningful conclusions from them. Further, we use the scipy module to help in calculating the different utility measures. Finally, we make use of the matplotlib module to plot our top interesting views.

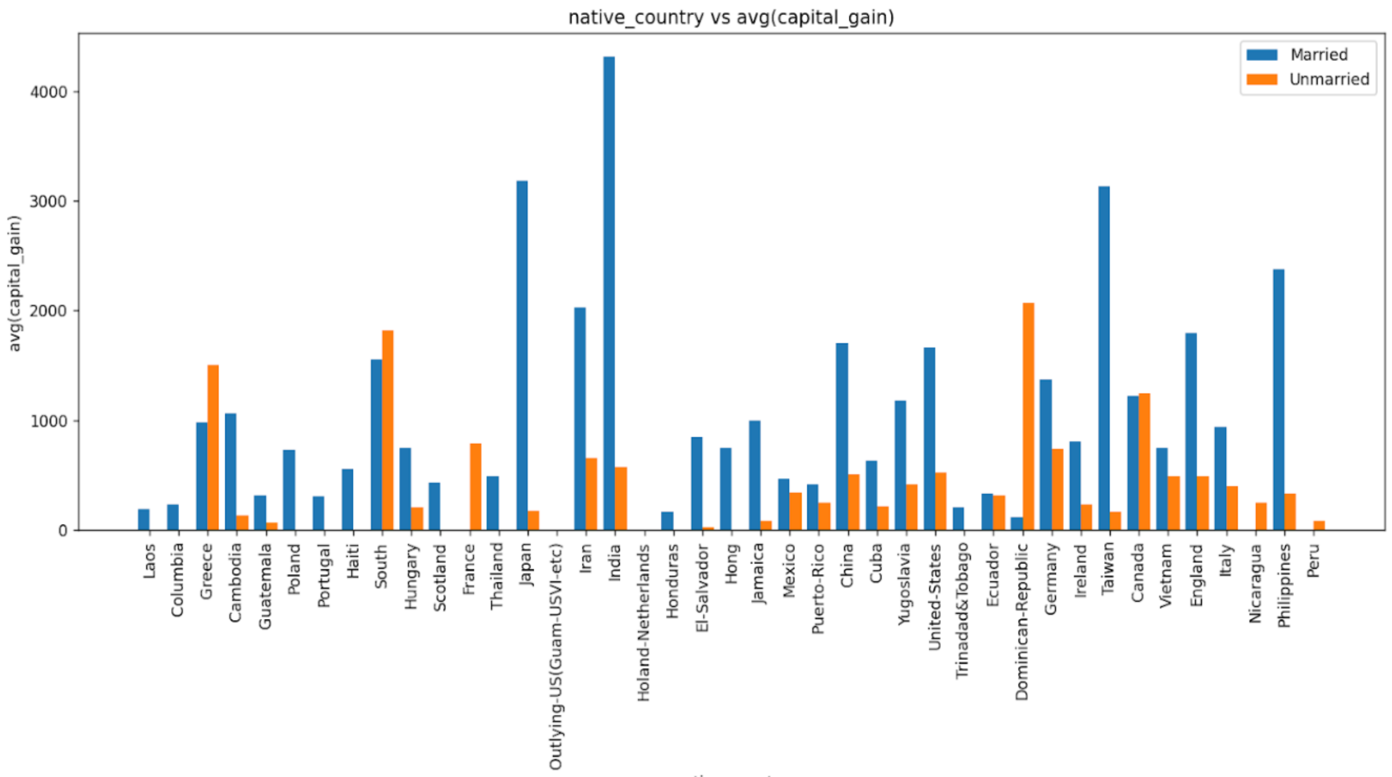
Experimental results

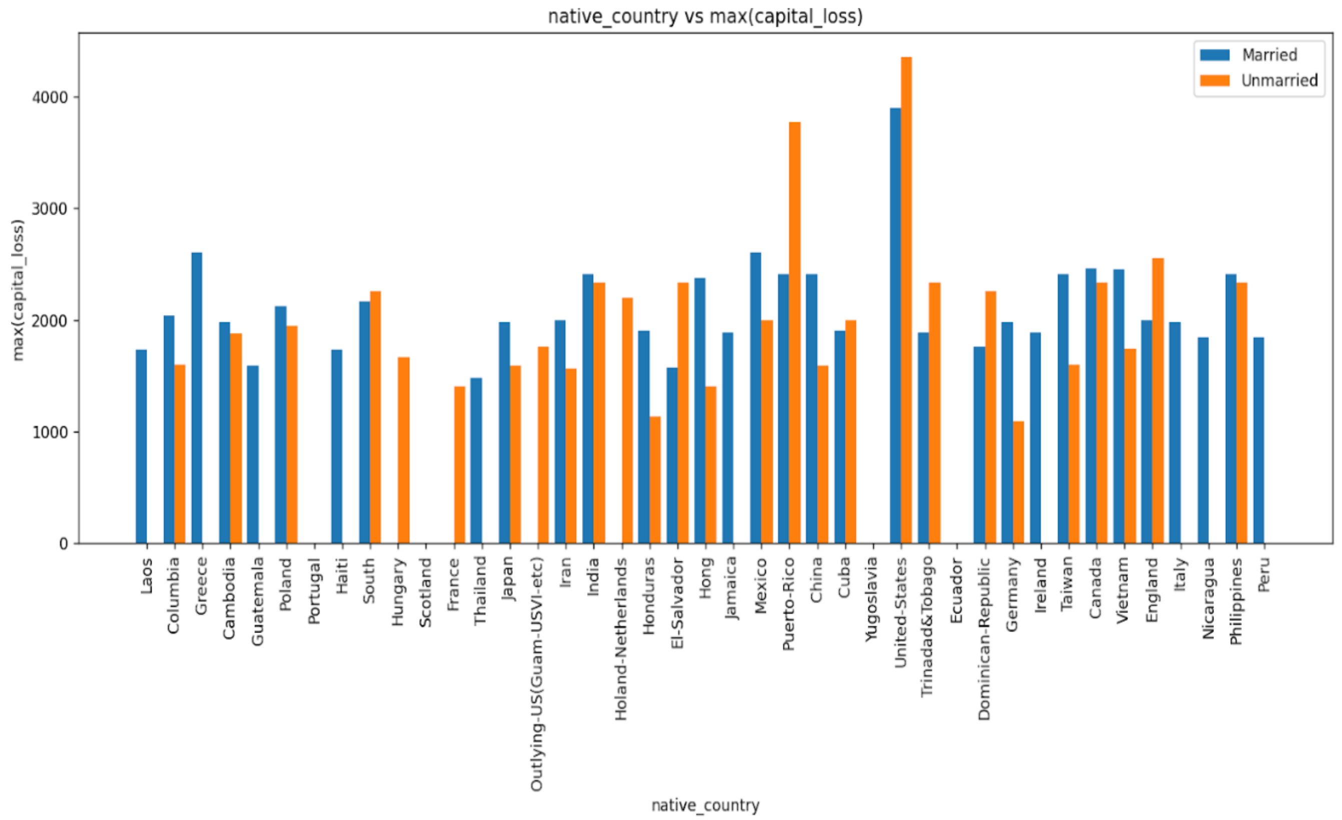
In this section, we discuss how well our experimental results show that the proposed techniques solve the problem presented at the beginning of the paper.

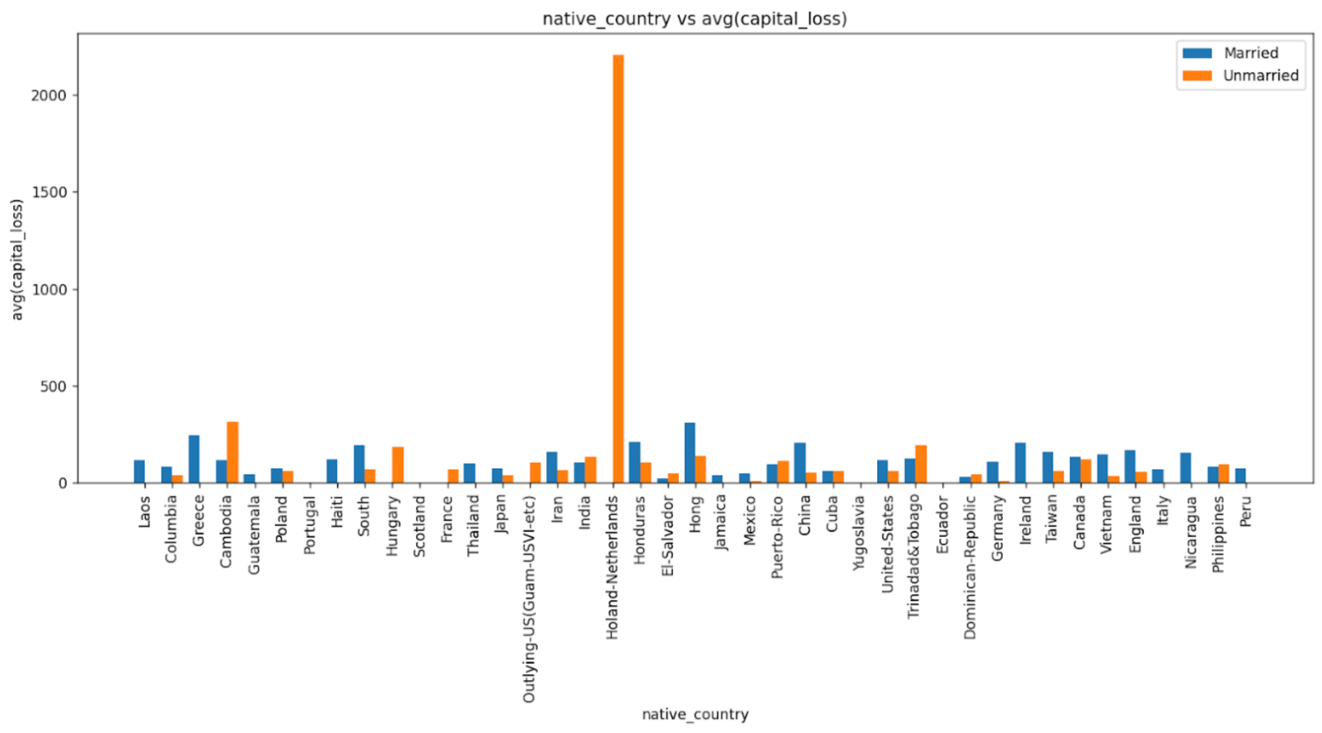
The top-5 interesting visualizations that our implementation of SeeDB produced are as follows:


workclass vs avg(capital_loss)
Married
Unmarried
avg(capital_loss)
0
25
50
75
100
125
150
175
200
Federal-gov
State-gov
Self-emp-inc
Self-emp-not-inc
Without-pay
Private
Never-worked
Local-gov
workclass


relationship vs max(capital_loss)
Married
Unmarried
max(capital_loss)
0
1000
2000
3000
4000
Own-child
Other-relative
Husband
Wife
Unmarried
Not-in-family


native_country vs avg(capital_gain)
Married
Unmarried
avg(capital_gain)
native_country
Laos
Columbia
Greece
Cambodia
Guatemala
Poland
Portugal
Haiti
South
Hungary
Scotland
France
Thailand
Japan
Outlying-US(Guam-USVI-etc)
Iran
India
Holand-Netherlands
Honduras
El-Salvador
Hong
Jamaica
Mexico
Puerto-Rico
China
Cuba
Yugoslavia
United-States
Trinadad&Tobago
Ecuador
Dominican-Republic
Germany
Ireland
Taiwan
Canada
Vietnam
England
Italy
Nicaragua
Philippines
Peru


native_country vs max(capital_loss)
Married
Unmarried
max(capital_loss)
native_country
Laos
Columbia
Greece
Cambodia
Guatemala
Poland
Portugal
Haiti
South
Hungary
Scotland
France
Thailand
Japan
Outlying-US(Guam-USVI-etc)
Iran
India
Holand-Netherlands
Honduras
El-Salvador
Hong
Jamaica
Mexico
Puerto-Rico
China
Cuba
Yugoslavia
United-States
Trinadad&Tobago
Ecuador
Dominican-Republic
Germany
Ireland
Taiwan
Canada
Vietnam
England
Italy
Nicaragua
Philippines
Peru



Extension and Results

Extension 1 - Analyze the DBLP dataset to find interesting patterns

In addition to the SeeDb algorithm implementation and Census dataset analysis, we implemented a variation of the base algorithm (to fit the DBLP model) to analyze the DBLP dataset and generate a report of all the interesting patterns found in the data. The DBLP computer science bibliography is a dataset that provides a comprehensive list of research papers in computer science. The DBLP dataset is a particularly large dataset and analyzing it requires an aggregated relation (after joining the 4 individual relations) and several preprocessing steps.

Data Preprocessing

Our implementation for finding interesting patterns in the DBLP dataset required a lot of data preprocessing. Since the DBLP dataset didn't have many rows with quantitative values we created our own columns to find attribute measures to aggregate over. Additionally, DBLP has 4 relations and SeeDB operates over a single relation, hence we created our own custom relation. The DBLP dataset is extremely large so we filtered our case to find interesting recommendations only for papers related to ‘Database’.

The following were the preprocessing steps undertaken:

1. Creating the custom table:
    a. We joined the papers, authors, venue, and paperauths table and selected only the papers with the keyword ‘Database’.

       Query :

       ```
       create view joined_view as
       (select year, school, v.name as venue, a.name as author, p.pages, p.name as title,
       ```

```
 p.id as paperid, v.type as venue_type
from papers p, authors a, venue v, paperauths pa
where a.id = pa.authid and p.id = pa.paperid and v.id = p.venue and
(p.name like '%Database%' or  p.name like '%database%'));
```

b. We wanted a new measure to count the number of co-authors for each paper. Hence we created a new view for coauthors count.
   Query:
   ```
   create view author_count as
   (select count(a.name), p.name, p.id as paperid
   from papers p, authors a, paperauths pa
   where a.id = pa.authid and p.id = pa.paperid
   group by p.id);
   ```
c. We now needed a combined table from the previous 2 views that we could feed into SeeDB.
   Query:
   ```
   create view final_view as
   (select year, school, venue, j.author, j.pages, a.name as title,
   a.count as coauthors, venue_type
   from joined_view j, author_count a
   where j.paperid = a.paperid);
   ```

2. We make use of 2 measures for our analysis coauthors and pages. We have calculated coauthors from our second query above but the page count attribute is in the form of a string (Eg. 3-10, 6:1-6:15, 5,10). We clean up this column in the table using the *clean_dblp_script* that calculates the page count for each of the papers as an integer so that they can be used as a measure for the aggregation functions.
3. Lastly, most of the rows have an empty value for the school attribute as those papers were not published by students or faculty belonging to educational institutions. We replace all such values with a new attribute 'Not from School'.

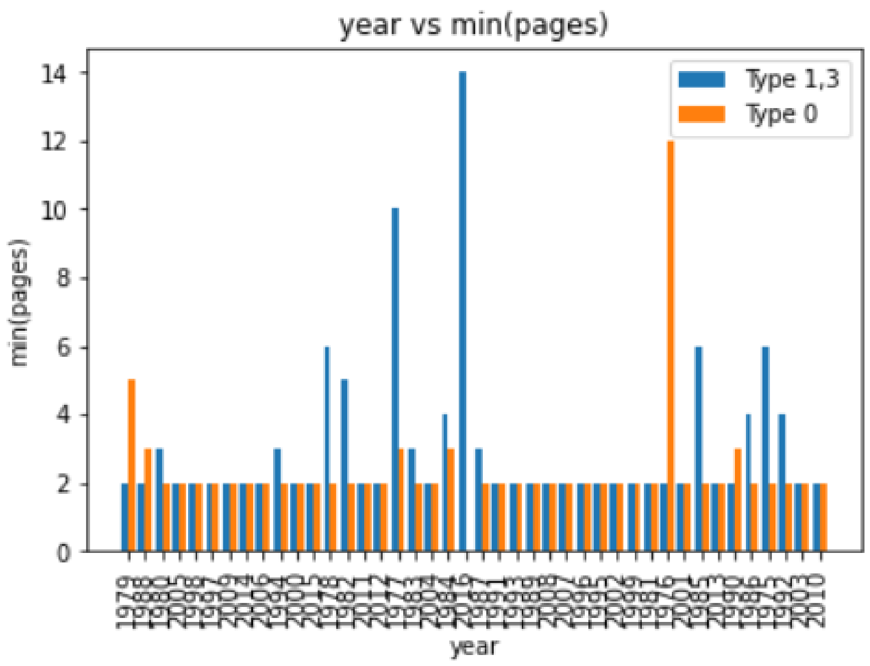
Implementation

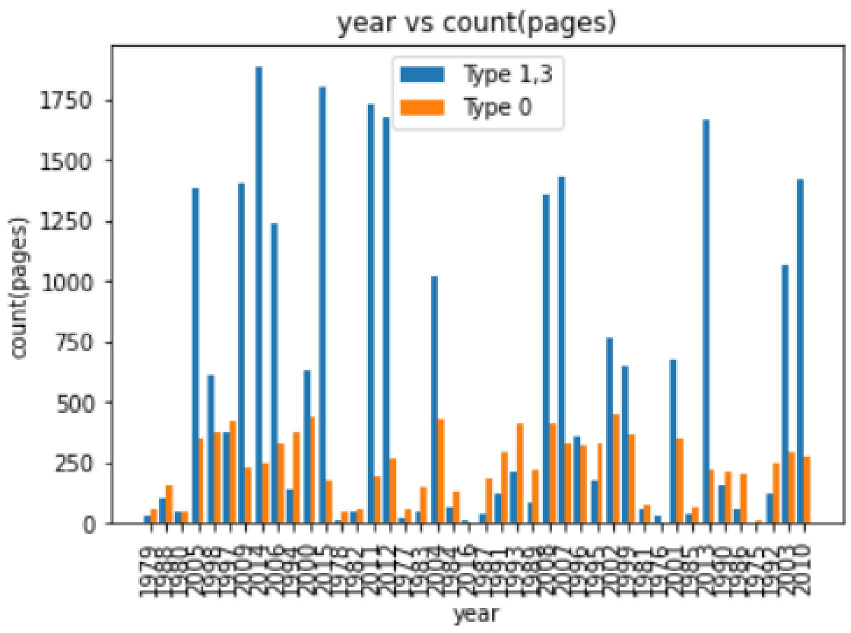
In our implementation, we set the values of all parameters from the problem statement as follows:

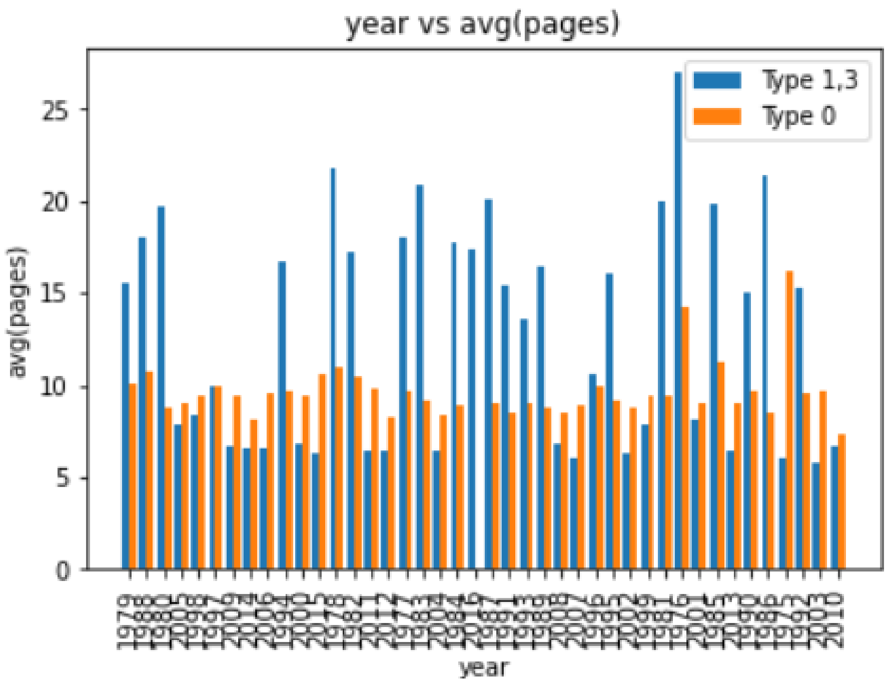
Parameter	**Value**
Dataset	DBLP
Target Dataset	Journal Articles venues (type 0)
Reference Dataset	All other venues (Conference and Workshop Pa oks and Thesis)
Dimension Attributes	year, school, venue
Measure Attribute	pages, coauthors
Aggregation Attribute	sum, min, max, avg, count
Number of splits	5

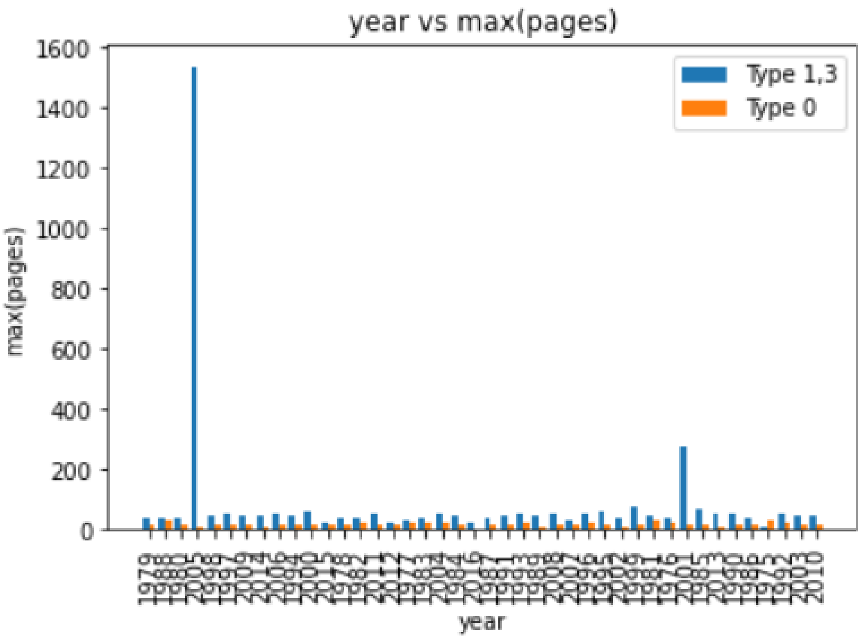
k	5

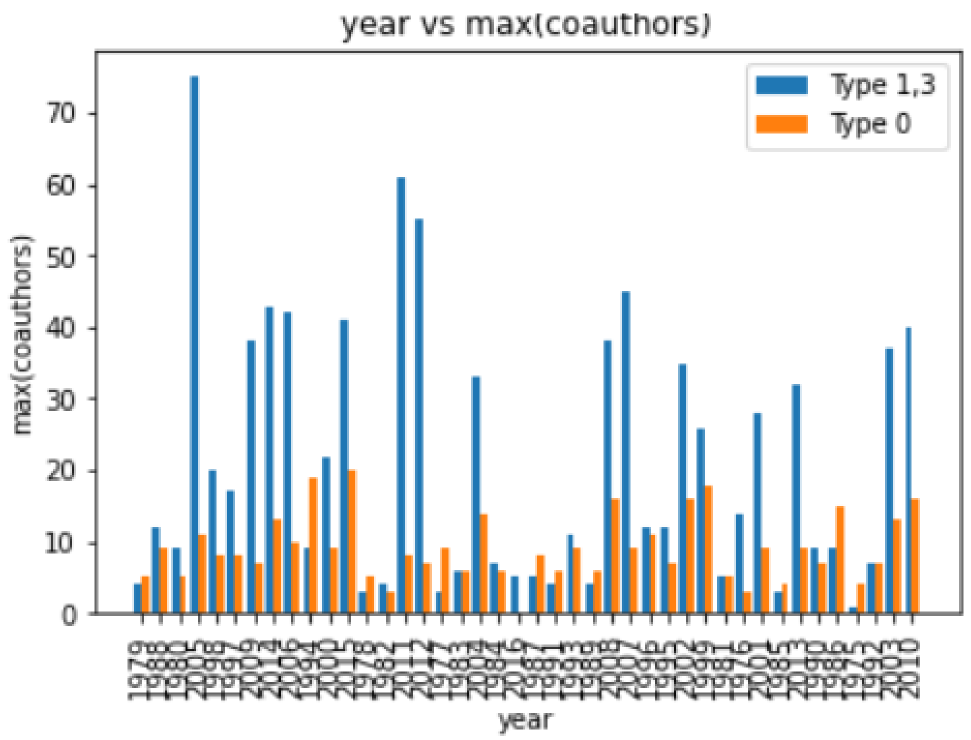
Results
We obtained the following top visualizations for the DBLP dataset.






year vs avg(pages)
Type 1,3
Type 0
avg(pages)
year


year vs max(pages)
Type 1,3
Type 0
max(pages)
year



Extension 2 - Query Rewriting

As a part of this extension, we implement query rewriting, an improvement over the base sharing-based optimization. Since the target and reference query are essentially the same and only operate on different subsets of the original dataset, we rewrite both queries as a single query using nuanced SQL semantics. This extension helps us further reduce the number of SQL queries fired and proves especially useful when the possible aggregate views are large. We achieved a significant speedup post the implementation of query rewriting.

Extension 3 - Use of additional utility measures

In addition to the stipulated KL Divergence deviation metric, we will also make our implementation flexible enough to compute the visualizations for several other utility measures.

We implemented the following utility measures:

1. **Earth Mover's distance:** EMD is a measure of the distance between two probability distributions over a region and is also known as the Wasserstein metric.
2. **Euclidean distance:** The Euclidean distance between two points in Euclidean space is the length of a line segment between the two points.
3. **Jensen–Shannon divergence:** JS Divergence is a method of measuring the similarity between two probability distributions and is also known as information radius or total divergence to the average.

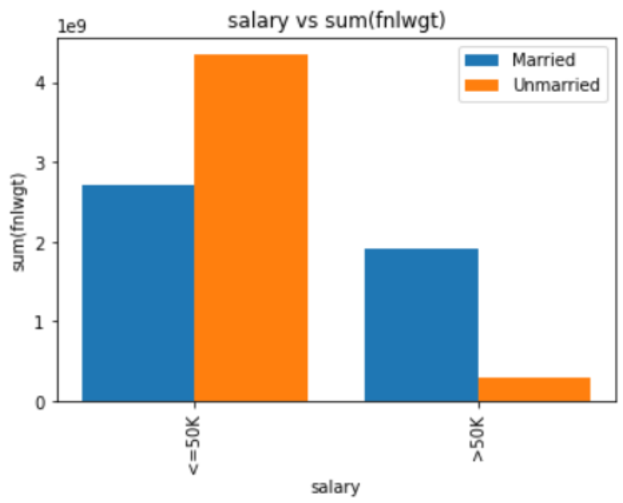
We noticed that using different utility measures resulted in some differences in the top k views produced, although not all of them varied.

We analyzed these results and came to the conclusion that albeit being different, these views were equally interesting as they highlighted significant differences in the behaviors of certain attributes in the target in comparison to the reference.

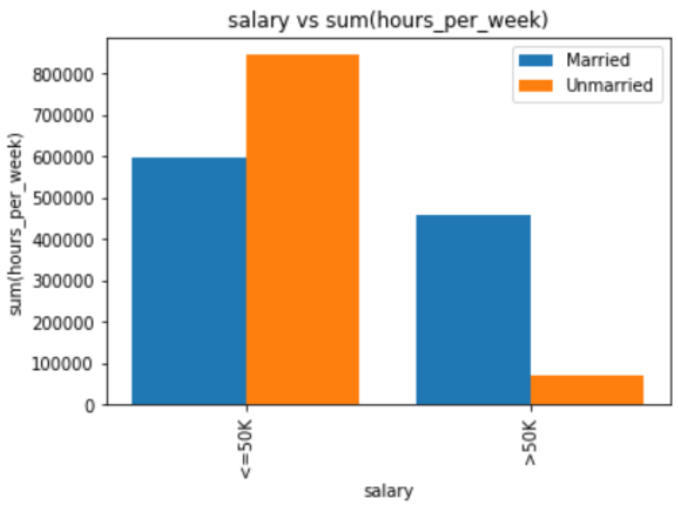
Results

The following represent some visualizations obtained upon using Earth Mover’s Distance as a utility measure.

```
Visualizing salary v/s sum(fnlwgt)
```

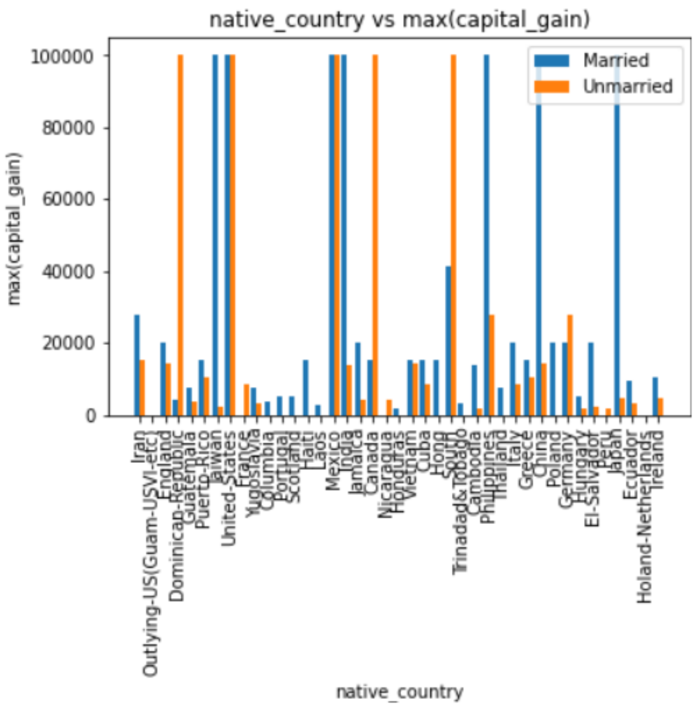


```
Visualizing salary v/s sum(hours_per_week)
```

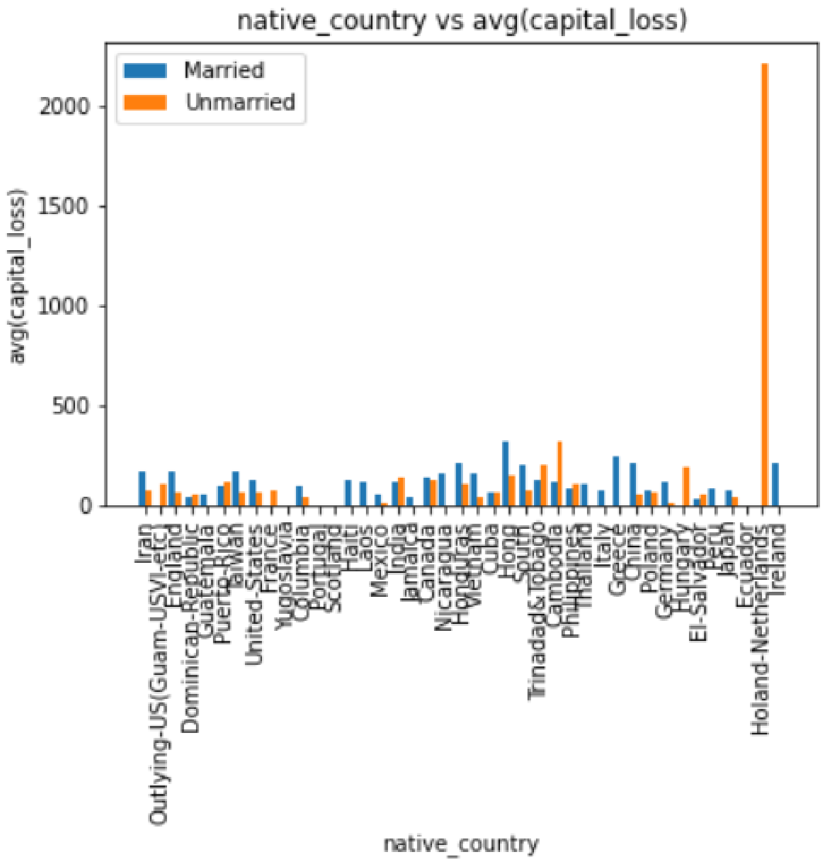


The following represent some visualizations obtained upon using JS Divergence as a utility measure.

```
Visualizing native_country v/s max(capital_gain)
```

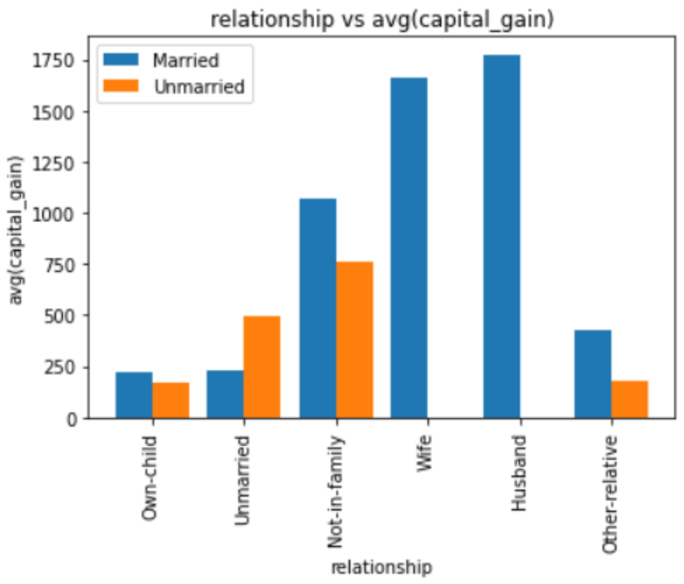


```
Visualizing native_country v/s avg(capital_loss)
```

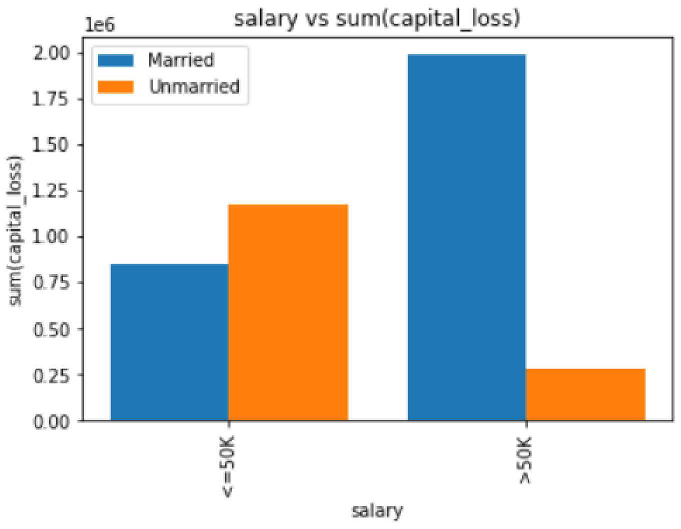


The following represent some visualizations obtained upon using Euclidean Distance as a utility measure.

```
Visualizing relationship v/s avg(capital_gain)
```



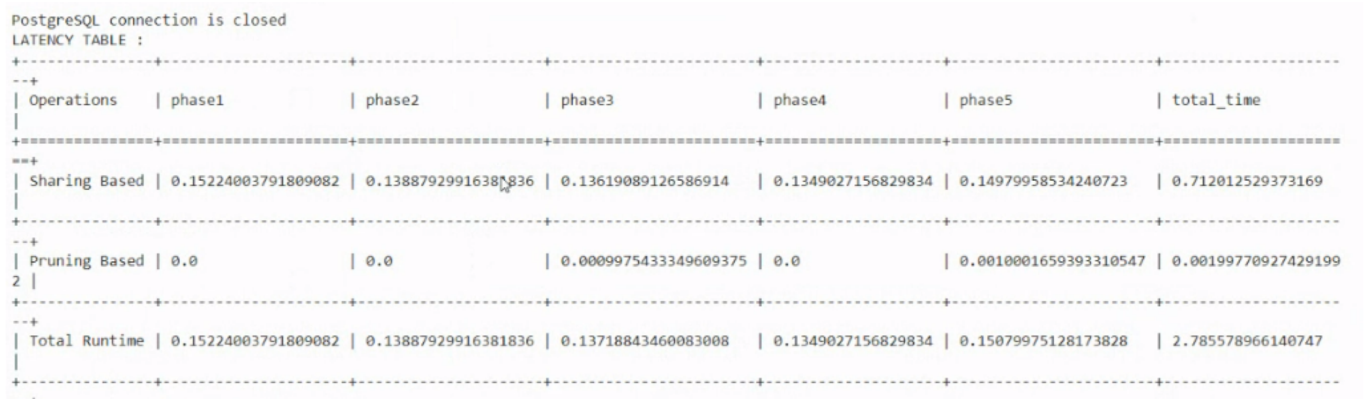
```
Visualizing salary v/s sum(capital_loss)
```



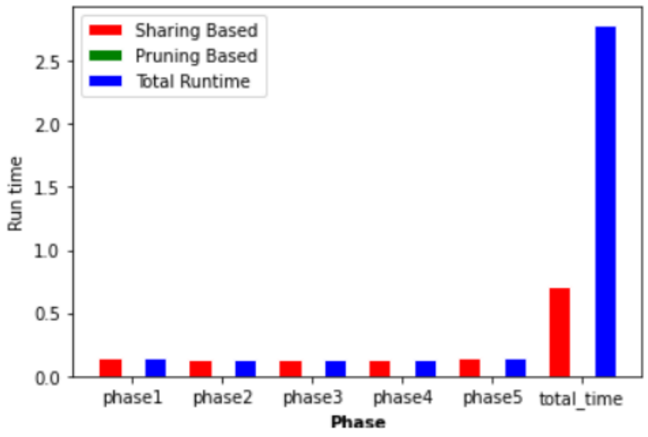
Summary

To summarize, the experimental results obtained show that the proposed techniques solve the problem presented at the beginning of the paper very well. The overall performance of SEEDB for two real data sets, Census and DBLP are captured in our project. This includes implementations for sharing and pruning-based optimizations. We have also developed three extensions, namely, query rewriting (an improvement on sharing base optimization), extended our architecture for a data set (DBLP) other than Census, and included four utility measures. Furthermore, to evaluate the results obtained, we designed an algorithm to compare the latencies for sharing-based optimization and pruning-based optimization for different utility measures. The following captures the table and latency plots for KL Divergence and Earth Mover's Distance. Looking at the tables and latency plots, we can see that Pruning based optimizations take much less time compared to Sharing Based Optimization.

KL Divergence latency distribution

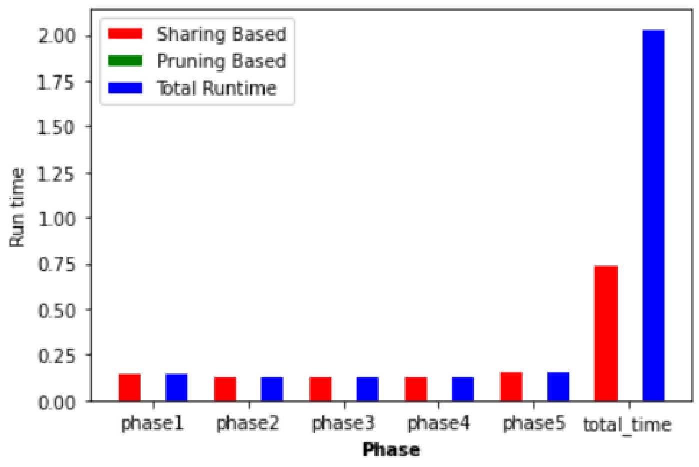


```
PostgreSQL connection is closed
LATENCY TABLE :
+---------------+---------------------+---------------------+-----------------------+---------------------+-----------------------+--------------------
--+
| Operations    | phase1              | phase2              | phase3                | phase4              | phase5                | total_time
|
+===============+=====================+=====================+=======================+=====================+=======================+====================
==+
| Sharing Based | 0.15224003791809082 | 0.13887929916381836 | 0.13619089126586914   | 0.1349027156829834 | 0.14979958534240723   | 0.712012529373169
|
+---------------+---------------------+---------------------+-----------------------+---------------------+-----------------------+--------------------
--+
| Pruning Based | 0.0                 | 0.0                 | 0.0009975433349609375 | 0.0                 | 0.0010001659393310547 | 0.00199770927429199
2 |
+---------------+---------------------+---------------------+-----------------------+---------------------+-----------------------+--------------------
--+
| Total Runtime | 0.15224003791809082 | 0.13887929916381836 | 0.13718843460083008   | 0.1349027156829834 | 0.15079975128173828   | 2.785578966140747
|
+---------------+---------------------+---------------------+-----------------------+---------------------+-----------------------+--------------------
```



Earth Mover's Distance latency distribution

```
LATENCY TABLE :
+---------------+---------------------+---------------------+---------------------+-----------------------+-----------------------+--------------------
---+
| Operations    | phase1              | phase2              | phase3              | phase4                | phase5                | total_time
|
+===============+=====================+=====================+=====================+=======================+=======================+====================
===+
| Sharing Based | 0.15813541412353516 | 0.13802003860473633 | 0.14083409309387207 | 0.13878893852233887   | 0.16895246505737305   | 0.7447309494018555
|
+---------------+---------------------+---------------------+---------------------+-----------------------+-----------------------+--------------------
---+
| Pruning Based | 0.0                 | 0.0                 | 0.0                 | 0.0010001659393310547 | 0.0009274482727050781 | 0.00192761421203613
28 |
+---------------+---------------------+---------------------+---------------------+-----------------------+-----------------------+--------------------
---+
| Total Runtime | 0.15813541412353516 | 0.13802003860473633 | 0.14083409309387207 | 0.13978910446166992   | 0.16987991333007812   | 2.037055730819702
|
+---------------+---------------------+---------------------+---------------------+-----------------------+-----------------------+--------------------
```



Section VI: Team Contribution

Teamwork was an important part of our project. All team members discussed and contributed to the design and initial implementation of the project. Once the design and initial implementation were completed, we listed out specific deliverables for each person and ensured that tasks could be done in parallel. In particular, Divya was responsible for the initial census database setup, while Neha was responsible for the initial DB setup of the DBLP dataset. This involved creating the relations, and views and importing the data. Chirag was responsible for cleaning up and preprocessing both the Census and DBLP datasets. Chirag and Neha were also responsible for implementing the Postgres database connectors, data partitioning logic, and visualization. Divya and Chirag were also responsible for implementing all the utility measures while Divya and Neha worked on the query generator, query utils, and the cleanup utils. Finally, all three members jointly worked on the core SeeDb logic (including optimizations) for both datasets. We have included the names of people who contributed to each function/file in the source code as comments. Further, all members of the group worked on documentation including Report and README. Similarly, all the inline documentation (comments) was done by all members of the team.