

Turbocharging Geospatial Visualization Dashboards via a Materialized Sampling Cube Approach

Jia Yu
Arizona State University
699 S. Mill Avenue, Tempe, AZ
jiayu2@asu.edu

Mohamed Sarwat
Arizona State University
699 S. Mill Avenue, Tempe, AZ
msarwat@asu.edu

Abstract—In this paper, we present a middleware framework that runs on top of a SQL data system with the purpose of increasing the interactivity of geospatial visualization dashboards. The proposed system adopts a sampling cube approach that stores pre-materialized spatial samples and allows users to define their own accuracy loss function such that the produced samples can be used for various user-defined visualization tasks. The system ensures that the difference between the sample fed into the visualization dashboard and the raw query answer never exceeds the user-specified loss threshold. To reduce the number of cells in the sampling cube and hence mitigate the initialization time and memory utilization, the system employs two main strategies: (1) a partially materialized cube to only materialize local samples of those queries for which the global sample (the sample drawn from the entire dataset) exceeds the required accuracy loss threshold. (2) a sample selection technique that finds similarities between different local samples and only persists a few representative samples. Based on the extensive experimental evaluation, Tabula can bring down the total data-to-visualization time (including both data-system and visualization times) of a heat map generated over 700 million taxi rides to 600 milliseconds with 250 meters user-defined accuracy loss. Besides, Tabula costs up to two orders of magnitude less memory footprint (e.g., only 800 MB for the running example) and one order of magnitude less initialization time than the fully materialized sampling cube.

I. INTRODUCTION

When a user explores a spatial dataset using a visualization dashboard, such as Tableau and ArcGIS, that often involves several interactions between the dashboard and the underlying data system. In each interaction, the dashboard application first issues a query to extract the data of interest from the underlying data system (e.g., PostGIS and Apache Spark SQL), and then runs the visual analysis task (e.g., heat maps and statistical analysis) on the selected data. Based on the visualization result, the user may iteratively go through such steps several times to explore various subsets of the database.

Running example. Figure 1 depicts a dashboard that visualizes 700 Million taxi rides data stored as a 100 GB database table; each tuple represents a taxi ride with various attributes (i.e., columns) such as the pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances (denoted as **D**), passenger count (denoted as **C**), payment method (denoted as **M**), itemized fare amount, tip and so on. The user can first select all taxi rides paid by cash using filters on the right pane and then plots the pick-up location of such rides on a heat map. She may then select taxi rides paid by credit card and render

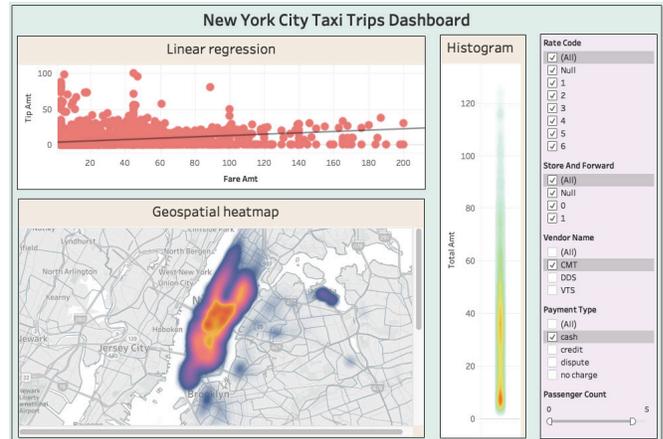


Fig. 1: A real interactive spatial visualization dashboard in Tableau

another heat map to visually compare the difference between the two maps.

Every interaction between the visualization dashboard and the underlying data system may take a significant amount of time (denoted as data-to-visualization time) to run, especially over large-scale data. The reason is two-fold: (1) The data-system query time proportionally increases with the volume of the underlying data table. Even scalable data processing systems such as Apache Spark and Hadoop, which parallelize the query execution, still exhibit non-negligible latency on large scale data. (2) Existing spatial visualization dashboards such as Tableau, ArcGIS and Apache Zeppelin work well for small to medium size data but do not scale to large datasets. Furthermore, since the user may perform various visualization effects on the same dashboard (e.g., 3 different tasks in Figure 1), practitioners would prefer to use a more generic approach to reduce the data-to-visualization time rather than install several different and isolated systems.

To remedy that, one approach that practitioners use is to draw a smaller sample of the entire data table (e.g., 1 million tuples) and materialize the sample in the database. This approach then keeps executing the dashboard on the materialized sample instead of the actual data set. The caveat is that running queries on the sample may lead to inaccurate

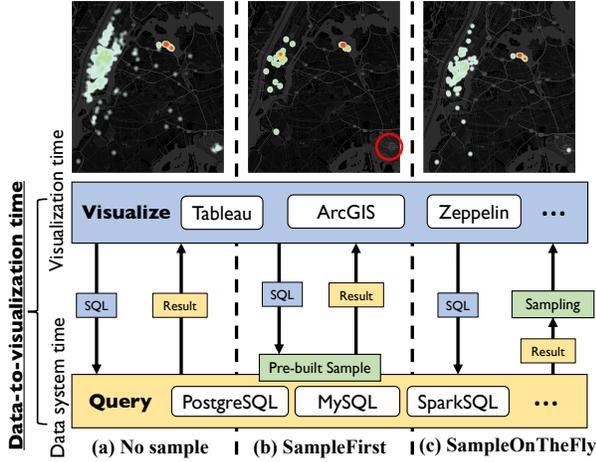


Fig. 2: Iterations between spatial visualization dashboards and data systems with different sampling approaches

visualization results since the query answer may significantly deviate from the actual answer especially for some small data populations. As shown in Figure 2, the approach that runs a query on the pre-built sample (denoted as *SampleFirst*) generates different visualization results in Tableau (Figure 2b) as compared to the approach that runs the query on the entire data table (Figure 2a). The *SampleFirst* approach even misses important visual patterns (the taxi rides from an airport, the red circle), and hence may mislead the user.

Recent research works such as Sample+Seek [1], BlinkDB [2], and SnappyData [3] address the problem of enhancing the accuracy of pre-built samples for approximate query processing. These approaches create stratified samples over multi-dimensional data to improve accuracy with a given confidence level. However, the pre-built stratified samples have no **deterministic** accuracy guarantee. So these systems may still need to perform some queries over the entire underlying table in an online fashion. Most importantly, all aforementioned approaches only support classic OLAP aggregate measures, such as COUNT, AVG, and cannot be easily extended to other types of data analysis (e.g., linear regression and most spatial visual effects in Figure 1). Instead of creating pre-built samples, an alternative approach runs data-system queries over the entire table for every iteration, draws a sample of the extracted population and sends it back to the visualization dashboard to shorten the visualization time. Although this approach (denoted as *SampleOnTheFly*) can certainly achieve higher and deterministic accuracy for the selected population, it is prohibitively expensive since it has to query the original table to prepare the sample for every user interaction.

In this paper, we present Tabula, a middleware framework that sits between a SQL data system and a spatial visualization dashboard to make the user experience with the dashboard more seamless and interactive. Tabula can seamlessly integrate

with the existing data system infrastructure (e.g., PostgreSQL, SparkSQL). As opposed to Nanocube and its variants [4], Tabula (given its inherent design as a middleware system) can work in concert with existing visual exploration tools such as Tableau and ArcGIS. Similar to Tabula, POIsam [5] and VAS [5] propose an online sampling technique to produce samples specifically optimized for spatial visual analysis. However, as opposed to Tabula, POIsam and VAS resort to the SampleOnTheFly approach to guarantee the sampling quality, which takes its toll on the overall data-to-visualization (as we prove in Section V).

Tabula adopts a materialized sampling cube approach, which pre-materializes samples, not for the entire table as in the SampleFirst approach, but for the results of potentially unforeseen queries (represented by an OLAP cube cell). Note that Tabula stores the sampling cube in the underlying data system. In each dashboard interaction, the system fetches a readily materialized sample for a given SQL query, which mitigates the data-system time. To scale, Tabula employs two strategies to reduce the sampling cube initialization time and memory utilization: (1) a partially materialized cube which only materializes local samples of those queries for which the global sample (the sample drawn from the entire dataset) exceeds the required accuracy loss threshold. (2) a sample selection technique to further reduce memory footprint. It finds similarities between different local samples, only persists a few representative samples, then uses the representative sample as an answer to many queries.

Since the dashboard application may show several types of visualization effects (see Figure 1), Tabula allows users to extend the system’s functionality by declaring their own user-defined accuracy loss function that fits each specific visualization effect. The system automatically incorporates the user-defined accuracy loss function in the sampling cube initialization and representative sample selection algorithms. Moreover, it always ensures that the accuracy loss due to using the sample never exceeds a user-specified deterministic accuracy loss threshold (100% confidence). That happens because Tabula efficiently examines the accuracy loss for all unforeseen queries when initializing the sampling cube.

We built a prototype of Tabula on top of SparkSQL and conducted extensive experiments to study the performance of Tabula and several other systems such as SampleFirst, SampleOnTheFly, SnappyData [3] and POIsam [5]. Based on the experiments, Tabula can bring down the total data-to-visualization time (including both data-system and visualization times) of a heat map generated over 700 million taxi rides to 600 milliseconds with 250 meters user-defined accuracy loss. It could be up to 20 times faster than its counterparts. Besides, Tabula costs up to two orders of magnitude less memory footprint (e.g., only 800 MB for the running example) and one order of magnitude less initialization time than the fully materialized sampling cube approach.

It is worth noting that the techniques proposed in this paper may be applied to both geospatial data and regular data visual analysis. For example, Section II shows that the generic user-

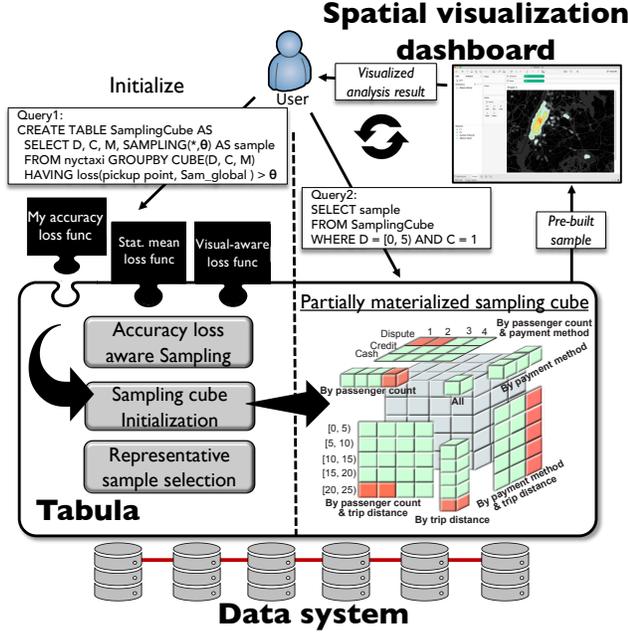


Fig. 3: Tabula overview. Samples in red cells are materialized. DCM cuboid doesn't show due to visualization limitation.

defined accuracy loss function can be about statistical mean and geospatial heat maps. Having said that, we believe that geospatial visualization is the most important scenario on which Tabula has a direct impact.

II. USING TABULA

Figure 3 gives an overview of Tabula. A user must initialize Tabula by providing the following system parameters as input: (1) User-defined accuracy loss function (abbr. $loss()$): This function determines how to calculate the accuracy loss due to using the sample as opposed to the original query answer. (2) Accuracy loss threshold θ : this parameter decides the acceptable accuracy for all queries processed by Tabula (3) Target attribute $attr$ on which $loss()$ measures the accuracy loss (4) Cubed attributes: the set of attributes that will be used to build the sampling cube (e.g., attributes D , C and M). Data-system SQL queries will use these attributes in WHERE clause predicates. The user feeds such parameters to Tabula as follows:

```
CREATE TABLE [sampling cube name] AS
SELECT [cubed attributes], SAMPLING(*, [θ]) AS sample
FROM [table name]
GROUPBY CUBE([cubed attributes])
HAVING [loss function name]([attr], Samglobal) > [θ]
```

where Sam_{global} represents a sample constructed by Tabula over the entire table using random sampling. $SAMPLING()$ is a Tabula-specific function that takes a dataset represented as a set of tuples and produces a sample of that dataset such that the accuracy of the produced sample, compared to the original dataset, does not exceed the accuracy loss threshold θ . Using the above SQL query, Tabula leverages the underlying data system to initialize a materialized sampling cube. The data

system can be any system that supports the CUBE operator. Query1 in Figure 3 is an initialization query for the running example.

Once the sampling cube is initialized, the user, via the spatial visualization dashboard, can issue SQL queries to Tabula, as follows:

```
SELECT sample FROM [sampling cube name]
WHERE [conditions]
```

After receiving this query, Tabula directly fetches a materialized sample from the sampling cube and returns it back to the visualization dashboard. This way, Tabula significantly reduces both the data-system time and visualization time. Besides, the system always guarantees with 100% confidence level that the accuracy loss from using the returned sample, as compared to the original query answer, does not exceed the accuracy loss threshold θ . It is worth noting that the attributes in the WHERE clause must be a subset of the cubed attributes specified in the initialization query. Query2 in Figure 3 is an example query which asks for a sample of tuples that satisfy $D = [0, 5)$ AND $C = 1$. The user can then issue subsequent queries with a different set of attributes to run the same or different analysis on various populations.

User-defined accuracy loss function. The visual analytics result obtained from a sample should be very close to that from the raw data. In this paper, we formalize the difference as accuracy loss. There are many ways to compute accuracy loss, which serve different purposes. In fact, one size does not fit all. The accuracy loss highly depends on the type of analysis the user plans to perform. That is the main reason why Tabula provides a generic approach for the user to declaratively define her custom made accuracy loss function that suits the analytics task at hand. The body of this function is a user-defined scalar expression over several aggregate functions. The standard SQL syntax is given below (supported by databases such as PostgreSQL):

```
CREATE AGGREGATE loss(Raw, Sam)
RETURN decimal_value AS
BEGIN scalar_expression END
```

Such a function takes raw data and sample data as input, then returns a decimal value which is the accuracy loss. For instance, consider an analytics task which requires a low relative error between the statistical mean of the sample and the statistical mean of the raw data. Such an accuracy loss function is implemented in Tabula as follows:

Function 1. $BEGIN \quad ABS\left(\frac{AVG(Raw) - AVG(Sam)}{AVG(Raw)}\right) \quad END$

Another example is the geospatial heat map on taxi pickup locations in Figure 1, where the accuracy loss function can stem from recent work on spatial visualization-aware sampling (VAS [6] and POIsam [5]). In that case, the user may implement the accuracy loss as the average minimum distance between the sample and raw data, as follows:

Function 2. $BEGIN \quad \frac{1}{|Raw|} \sum_{x \in Raw} MIN_{s \in Sam}(loss_{pair}(x, s)) \quad END$

where $loss_{pair}(x, s)$ is the Euclidean distance, Manhattan distance or any distance metric between two data objects.

The third example is the linear regression analysis on trip tip amount VS. fare amount in Figure 1, where the accuracy loss function calculates the angle difference between the regression lines of raw data and sample data, as follows:

Function 3. *BEGIN ABS(angle(Raw) - angle(Sam)) END*

Given n tuples each of which has a 2D attribute (x_i, y_i) , we use the following function to calculate the slope [7]:

$$\text{slope} = \frac{n\sum(x_i * y_i) - \sum x_i * \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}$$

We then convert slope to angle (unit: degree $^\circ$). Eventually, the corresponding visual analysis task plots the regression line of data of interest: $y = \text{slope} * x + \text{intercept}$. In this example, the loss function uses fare amount as x , tip amount as y .

Tabula requires that the accuracy loss function must be algebraic (see definitions in Section VI). To achieve that, all aggregate functions and mathematical operators involved in calculating $\text{loss}(\text{Raw}, \text{Sam})$ must be distributive or algebraic. In fact, many common aggregations satisfy this restriction [8] including SUM, COUNT, AVG, STD_DEV, MIN, MAX, DISTINCT, TOP-K, excluding MEDIAN.

Once the user defines the accuracy loss function, Tabula embeds such a function in the core components of the sampling cube. For instance, if the user defines the statistical mean-aware accuracy loss function (discussed previously) and sets the value of the accuracy loss threshold $\theta = 10\%$, Tabula will guarantee with 100% confidence that the relative error due to using the statistical mean of every sample in the cube will never exceed 10%. On the other hand, if the user uses geospatial heat map-aware sampling accuracy loss and sets the value of θ to an absolute loss value, 1 meter, then Tabula guarantees that the average min distance between the raw query result and the returned sample will never exceed 1 meter.

III. MATERIALIZED SAMPLING CUBE

After the user issues the initialization query (presented in Section II), Tabula builds a *partially* materialized sampling cube and stores it in the underlying data system. The system only materializes local samples for a selected set of cells in the sampling cube, namely iceberg cells. A cell that satisfies $\text{loss}(\text{cell data}, \text{Sam}_{\text{global}}) > \theta$ (SQL equivalent: $\text{loss}([\text{target attribute}], \text{Sam}_{\text{global}}) > \theta$) is called an iceberg cell. Otherwise, Tabula will use the global sample to answer a query corresponding to a non-iceberg cell. Figure 3 gives the layout of a sampling cube, which contains a cube table (see Figure 4(a)) and a sample table (see Figure 4(b)). All cells depicted in Figure 4(a) are iceberg cells. A cell in the materialized sampling cube is defined as $\langle a_1, a_2, \dots, a_n : \text{sample_id} \rangle$, where n is the number of cubed attributes. sample_id points to a sample in the sample table and many cells may share the same sample_ids because of Tabula’s optimization in Section IV. Cell $\langle [0, 5], \text{null}, \text{null} : 1 \rangle$ and $\langle [0, 5], 1, \text{credit} : 1 \rangle$ both share the same sample set whose id is 1. ‘null’ indicates *. In the rest of this section, we will first explain how the sampling module of Tabula can harness the

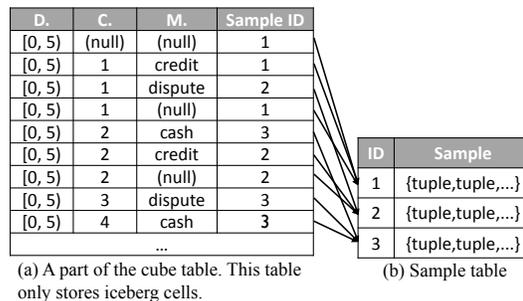


Fig. 4: Tabula sampling cube physical layout

user-defined accuracy loss function to draw samples. We will then explain how the system finds iceberg cells and efficiently constructs the sampling cube using the sampling module.

A. Accuracy Loss-Aware Sampling

The sampling function (i.e., $\text{SAMPLING}(*, [\theta])$ in Section III) aims at generating a sample with the objective to minimize the sample size while guaranteeing $\text{loss}(\text{Raw}, \text{Sam}) \leq \theta$. Since classic sampling algorithms such as random and stratified sampling do not handle a user-defined accuracy loss function, the sampling module in Tabula employs a generic sampling algorithm, which works for a generic accuracy loss function. The sampling problem can be formally defined as follows:

Definition 4 (Sampling problem). *Given a dataset T , an accuracy loss function ($\text{loss}()$), and an accuracy loss threshold θ , select a subset t from T such that: (1) $\text{loss}(T, t) \leq \theta$ and (2) The size of t is minimized.*

The sampling module in Tabula employs a greedy algorithm similar to the algorithm proposed by POIsam [5]. However, our algorithm guarantees that $\text{loss}(T, t) \leq \theta$ but the sample size may not be minimal. Algorithm 1 depicts the major steps of this algorithm: it first creates an empty sample set t which has $\text{loss}(T, t) = \infty$. In each greedy selection round, for every remaining tuple tp in T , it computes $\text{loss}(\text{original}T, t + tp)$. Original T is the original raw dataset T . It always picks from T (without replacement) the tuple which has the minimum loss and adds it to t . This algorithm keeps picking tuples from T and adds them into t until $\text{loss}(\text{original}T, t) \leq \theta$. Tabula further accelerates the greedy algorithm using the lazy-forward strategy of POIsam (not shown here). The final complexity of each greedy round is $O(k * N)$ where N is the size of input data and k is much smaller than N .

Lemma III.1. *The sampling function $\text{SAMPLING}()$ is a holistic aggregate function.*

Proof. The definition of holistic functions is given in Section VI. This lemma can be proved by contradiction but the full proof is omitted for brevity. \square

Algorithm 1: Greedy algorithm for sampling

Data: A dataset T , $\text{loss}()$, θ
Result: A sample t

- 1 Create an empty list t ;
- 2 Create a copy of T *originalT*;
- 3 $\text{loss} = \infty$;
- 4 **while** $\text{loss} > \theta$ **do**
- 5 $\text{minTuple} = \text{NULL}$;
- 6 **foreach** *tuple tp in T do*
- 7 $\text{tp_loss} = \text{loss}(\text{originalT}, t+tp)$;
- 8 **if** $\text{tp_loss} < \text{loss}$ **then**
- 9 $\text{loss} = \text{tp_loss}$;
- 10 $\text{minTuple} = tp$
- 11 $t.\text{add}(\text{minTuple})$;
- 12 $T.\text{remove}(\text{minTuple})$;
- 13 **return** t

Any data cube with such a holistic aggregate function cannot leverage state-of-the-art cube construction approaches [9].

B. Sampling Cube Initialization

The straightforward way to initialize a sampling cube is as follows: First, Tabula draws a global random sample, called $\text{Sam}_{\text{global}}$, from the entire raw dataset. Second, it builds the sampling cube by running a set of GroupBy queries to calculate all cuboids in the cube (a cuboid [10] is a GroupBy query). This can be done via using the SQL CUBE operator in the underlying DBMS (see Query 1 in Figure 3). Given the grouped raw data of each cube cell, if applying the global sample to this cell satisfies the iceberg condition, i.e., $\text{loss}(\text{cell data}, \text{Sam}_{\text{global}}) > \theta$, Tabula will identify this cell as an iceberg cell and generate / materialize a local sample (denoted as $\text{Sam}_{\text{local}}$) for it.

However, the cost of using the classic CUBE operator to build the sampling cube increases exponentially with the number of cubed attributes. In Figure 1, each record may have five attributes (filters) and running the CUBE operator on these attributes requires $(2^5 - 1)$ GroupBy operations over the entire table. Tabula avoids that by learning which cuboid of the sampling cube really contains iceberg cells before actually building the cube, then all unnecessary GroupBys can be avoided. To achieve that, after drawing the global sample, the algorithm runs in two main stages, namely: Stage 1: Dry run for iceberg cell lookup and Stage 2: Real run for sampling cube construction.

1) *Dry Run Stage: Iceberg Cell Lookup:* In this stage, the system identifies all iceberg cuboids (cuboids that have iceberg cells), by scanning the raw table data only once. According to the aforementioned straightforward initialization query, Tabula applies two aggregate functions to each iceberg cell in the sampling cube: $\text{SAMPLING}()$ and $\text{loss}()$. Based on the literature in OLAP data cube (see Section VI), we know that: if an aggregate measure is distributive or algebraic, existing algorithms only need to run the full table GroupBy operation once to build an initial cuboid and other cuboids can be built upon it. For a holistic aggregate measure, there are no better

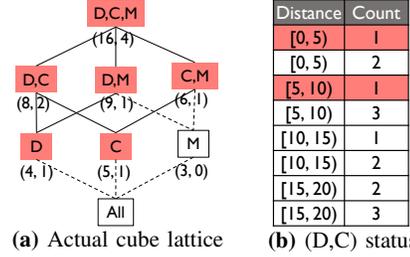


Fig. 5: Major steps in the initialization algorithm

algorithms to materialize the aggregate measure for each cell other than building every cuboid from the raw data [9].

Since the sampling function is holistic, although Tabula only materializes local samples for some cells of the cube (iceberg cells), the underlying data system has to run $(2^n - 1)$ full table GroupBy operations because it cannot speculate which cells are iceberg cells beforehand (n is the number of cubed attributes). However, since the loss function is algebraic, Tabula leverages such property in this stage by utilizing any existing cube initialization algorithms in Section VI to efficiently build a partially materialized sampling cube. Such a cube only uses accuracy loss as the aggregate measure. This way, Tabula only accesses the raw data once to build the top/bottom cuboid and then all other cuboids can be derived from the cuboid itself.

TABLE I: Example tables generated in the dry run stage

(a) Iceberg cell table			(b) Cuboid D,C,M		
D.	C.	M.	D.	C.	M.
(null)	(null)	credit	[0,5]	1	credit
[0, 5]	(null)	cash	[5,10]	1	credit
[5, 10]	(null)	(null)	[15,20]	2	cash
[0, 5]	1	(null)	[15,20]	3	cash
[5, 10]	1	(null)			
...					
(c) Cuboid D,C			(d) Cuboid D		
D.	C.	M.	D.	C.	M.
[0, 5]	1	(null)	[5, 10]	(null)	(null)
[5, 10]	1	(null)			

The output of the dry run stage is an iceberg cell table (Table Ia). Tabula then derives iceberg cell tables for each cuboid (e.g., Table Ib,Ic,Id). In addition, Tabula can also know the approximate number of all cells and iceberg cells in each cuboid by checking the global sample. Therefore, based on these outcomes, we can draw the lattice structure of Tabula (Figure 5a) without even computing any local samples for now. Each vertex of the lattice is a cuboid (i.e., GroupBy query) and letters indicate the attributes that appear on the grouping list of this cuboid. For instance, the top vertex DCM is the cuboid that has trip distance, passenger count and payment method. The bottom vertex “All” actually is not a GroupBy query because it has no attributes on the grouping list. Two cuboids 1 and 2 are connected by an edge only if the grouping list of cuboid 1 is a subset of the grouping list of cuboid 2. All cells in cuboid 1 can find their descendant cells in 2. As

Algorithm 2: Initialization: Real Run Stage

Data: The raw table *tbl* and results of the dry run stage

Result: A cube table including samples

```
1 foreach cuboid cbd in all cuboids do
2   if its iceberg cell table is not null then
3     if it satisfies Inequation 1 then
4       Run equality join tbl with the iceberg cells of
5         cbd to retrieve data;
6       Build cbd via a GroupBy on tbl or retrieved data;
7       Draw a local sample for each iceberg cell;
8   else
9     Skip this cuboid;
```

depicted in Figure 5a, every colored cuboid contains at least one iceberg cell. The first number indicates all cells and the second number indicates iceberg cells.

Global sample size. The size of a sample affects its accuracy loss. Since Tabula checks the global random sample against every single cube cell during the dry run stage (builds local samples later if necessary), the size of the global sample has no effect on Tabula’s error bound which is the loss threshold. However, a too small global sample may unnecessarily introduce too many iceberg cells. Therefore, Tabula utilizes Serfling’s Inequality [11], [5] (a lemma of the law of large numbers) to determine a proper global sample size. Let x_1, x_2, \dots, x_n be a finite set of numbers in $[0, 1]$ with a mean μ , for any $\epsilon > 0$ and $1 \leq k \leq n$, we have

$$\mathbb{P} \left[\max_{k \leq m \leq n-1} \left| \frac{1}{m} \sum_{i=1}^m x_i - \mu \right| \geq \epsilon \right] \leq 2 \exp \left(-\frac{2k\epsilon^2}{1 - \frac{k-1}{n}} \right) = \delta$$

where k is the sample size. Therefore, given any relative error ϵ of μ and confidence level δ , we have $k \approx \frac{\ln \frac{2}{2\epsilon^2}}{2\epsilon^2}$. By default, Tabula uses $\epsilon = 0.05$ and $\delta = 0.01$. Given the NYCtaxi dataset (700 million records) used in Section V, the global sample has around 1000 tuples. This makes sure that this sample can represent the distribution of the raw dataset.

2) *Real Run Stage: Sampling Cube Construction:* Based on the iceberg cell information learned in the dry run stage, Tabula constructs a sampling cube that only contains iceberg cuboids. For each cell in this cuboid, the algorithm draws a local sample if the cell is an iceberg cell. The algorithm performs the same step for all iceberg cuboids until it eventually builds the sampling cube (see Figure 6).

Algorithm 2 gives the detailed pseudocode of the real run stage. The dry run stage has shown the number of iceberg cells in each cuboid (e.g., Table Ia), so Tabula can easily skip these non-iceberg cuboids (uncolored cuboids in Figure 5a) and work on iceberg cuboids that have at least one iceberg cell (red-colored cuboids in Figure 5a). For each iceberg cuboid, the algorithm then fetches the raw data that correspond to each iceberg cell in this cuboid. That can be done in two different ways: (1) Run a GroupBy operation using the cuboid attributes on the raw data and check the iceberg condition before drawing the sample for a cell (2) Run an equi-join operation between the cuboid iceberg cell table and the raw data to find the data

D.	C.	M.	Cell raw data	Sample
[0, 5)	1	credit		
[0, 5)	1	dispute		
[0, 5)	2	(null)		
[0, 5)	2	cash		
...				

Fig. 6: Output cube table of the initialization algorithm

corresponding to iceberg cells (see Figure 5b), then run the GroupBy operation on the retrieved raw data of iceberg cells, and finally draw a local sample for such cells. The second way is obviously more efficient when the iceberg cuboid only has a few iceberg cells. To decide that, Tabula employs the following cost model:

$$\begin{aligned} Cost_{Prune} + Cost_{GroupPrunedData} &< Cost_{GroupAllData} \\ N * i + \frac{i}{k} N * \log_k \left(\frac{i}{k} N \right) &< N * \log_k(N) \end{aligned} \quad (1)$$

where N is the cardinality of the table, i is the number of iceberg cells, k is the number of all cells in this cuboid. If the inequality holds, Tabula will use the second way mentioned above. Note that this condition assumes that each cell has the same amount of grouped raw data.

IV. SAMPLE SELECTION

After the cube initialization, the partially materialized sampling cube may still possess a large memory footprint. That is because: (1) the number of cuboids and cube cells increase exponentially as the number of cubed attributes increase (2) for every iceberg cell, Tabula materializes a sample dataset (not just a single aggregate value), which may still consist of hundreds or thousands of tuples.

We observed that a sample in an iceberg cell can actually be re-used to represent the samples of other iceberg cells. That happens when applying the sample to those cells still ensures that $\text{loss}(\text{raw}, \text{sam}) \leq \text{threshold } \theta$. For example, in Figure 6, the sample stored in Iceberg Cell $\langle [0, 5), 1, \text{dispute} \rangle$ is similar to the raw data of Iceberg Cell $\langle [0, 5), 2, \text{null} \rangle$. In this case, we can let Cell $\langle [0, 5), 2, \text{null} \rangle$ use the sample of $\langle [0, 5), 1, \text{dispute} \rangle$ instead of materializing its own local sample. The sample of $\langle [0, 5), 1, \text{dispute} \rangle$ is the representative sample for these two iceberg cells’ samples. Therefore, to further reduce the memory footprint, Tabula only persists a representative set of samples from the cube table, and re-uses the representative samples in many iceberg cells rather

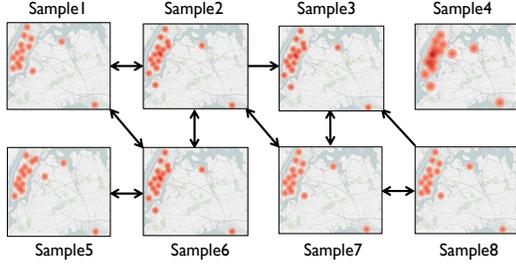


Fig. 7: Select the representative samples

than persisting every individual local sample. We define the representation relationship between two samples as follow:

Definition 5 (Sample Representation relationship). *Given the raw data of an iceberg cell $Cell_A$ (format: tuple, tuple, ...) and its local sample Sam_A (format: tuple, tuple, ...), the raw data of another iceberg cell $Cell_B$ and its sample Sam_B . Sam_A can represent Sam_B only if $loss(Cell_B, Sam_A) \leq loss\ threshold$.*

To select representative samples, Tabula first evaluates the relationships among different iceberg cells, which are described by a graph, namely sample representation graph (abbr. SamGraph). See the example in Figure 7.

Definition 6 (SamGraph). *$SamGraph(V, E)$ is a directed graph, where V and E represent the set of vertexes and edges, respectively. Each $v \in V$ represents a local sample stored in an iceberg cell. A directed edge from vertex v to u indicates that the sample v can represent the sample u . A bi-directed edge between a vertex v and u means that both samples v and u can represent each other.*

To build the SamGraph, Tabula performs an inner join on the cube table generated by the initialization algorithm (Section III-B). The join condition is the representation relationship depicted above. We can express the inner join query using a SQL query as follows:

```
SELECT t1.D, t1.C, t1.M, t2.D, t2.C, t2.M
FROM cube_table t1, cube_table_no_rawdata t2
WHERE loss(t1.cellrawdata, t2.sample) ≤ threshold
```

where `cube_table` is the cube table in Figure 6 and `cube_table_no_rawdata` is the same table but without raw data. Because the loss function may need some measures of the cell raw data (e.g., AVG), the cube table from the initialization algorithm needs to carry the raw data for each iceberg cell. Therefore, the cube table generated by the real run stage has a column named Cell Raw Data. Note that this join can be accelerated by any existing image/data similarity join algorithms. In addition, this join result does not have to exhaust all possible representation relationships. Sample selection on a non-exhaustive SamGraph may not minimize Tabula’s memory footprint but still ensures Tabula’s bounded-error guarantee.

Tabula traverses the SamGraph to select the representative samples, only persists selected samples, and drops the rest. We formally define the Representative Sample Selection (abbr. RepSamSel) problem as follows:

Algorithm 3: Representative Sample Selection

Data: $SamGraph(V, E)$, each edge is denoted as $\langle head, tail \rangle$. head and tail are sample IDs
Result: A set D which consists of many sample IDs

- 1 Group edges E by head sample IDs;
// Sort head sample IDs by their outdegree
- 2 Sort groups in the descending order of the group counts;
- 3 Create a LinkedHashMap $HM(\langle head, \{tail, tail, \dots\} \rangle)$;
- 4 Insert sorted groups one by one into HM ;
- 5 Create a representative set $D = \emptyset$;
- 6 **while** $HM \neq \emptyset$ **do**
// Pick sample ID by outdegree
7 Remove the top map $\langle head, \{tail, tail, \dots\} \rangle$ from HM ;
8 Put $head$ in D ;
// Remove samples that are represented by $head$
9 **foreach** $tail$ in $\{tail, tail, \dots\}$ **do**
10 | Remove the map whose key is $tail$, from HM

Definition 7 (Representative Sample Selection). *Given a SamGraph = (V, E) , select a subset D of V such that: (1) For every vertex $v \notin D$, v is represented by at least a vertex $u \in D$ and (2) The size of D is minimized.*

As depicted above, the main objective is to persist a minimal set of local samples and every unpersisted local sample can be represented by a persisted local sample. The first condition guarantees that if a local sample is not selected to be persisted, each iceberg cell can still use one of the persisted samples to answer queries. The second condition ensures that Tabula selects the minimum number of samples to persist/maintain, and hence reduces the overall memory space occupied by the sampling cube.

Lemma IV.1. *The representative sample selection (RepSamSel) problem is NP-Hard.*

Proof. That can be proved by reducing the Minimum Dominating Set (MDS) problem which is known to be NP-hard [12] to RepSamSel problem. We first relax the representation relationship: if sample A can represent sample B, then sample B can also represent A. Then, we can change all edges in SamGraph (V, G) to bi-directed edges. Now RepSamSel problem is identical to the MDS problem. The RepSamSel problem on bi-directed SamGraphs is a subset of that on directed SamGraphs. Therefore, RepSamSel problem is also NP-hard. The full details of the reduction algorithm is omitted due to space limitation. \square

Representative Sample Selection Algorithm. Since RepSamSel problem is NP-hard, we resort to a greedy selection strategy. The algorithm (see Algorithm 3) takes as input the SamGraph (V, E) and keeps selecting a sample $v \in V$ and inserts it into D based on a greedy strategy, until every remaining sample in V has at least one representative in D . The greedy strategy always picks the sample $v \in V$ such that v has the highest number of edges directed from v to other samples. In other words, the algorithm always selects the

most representative sample among all remaining samples in a greedy fashion. Given the SamGraph in Figure 7, the greedy representative sample selection algorithm will pick Sample2 (represents 1,2,3,6,7), Sample8 (represents 3,7,8), Sample5 (represents 5,6) and Sample4 (represents itself), in this particular order. These four samples compose the representative samples set and are persisted in a sample table as depicted in Figure 4(b). The resulting cube table is normalized to the final cube table such as Figure 4(a) and each iceberg cell of the final cube table links to a sample id. If the local sample of an iceberg cell can be linked to multiple samples, we randomly pick one link to keep.

V. EXPERIMENTS

Compared approaches. All pre-built samples are cached into the cluster’s memory: (1) SampleFirst (SamFirst): This approach creates a random sample of the entire dataset before accepting any query (see the definition in Section I). We use two SampleFirst versions: 100MB and 1GB pre-built sample sizes. (2) SampleOnTheFly (SamFly): This approach has no pre-built samples (see the definition in Section I). It uses the greedy sampling algorithm (Algorithm 1) to ensure the deterministic accuracy guarantee. (3) POIsam [5]: It is similar to SampleOnTheFly but has an extra random sampling step. After executing every query, it first creates a random sample on the query result then applies Algorithm 1. Please note that this greedy algorithm modifies the original POIsam’s algorithm which fixes returned sample size and minimizes accuracy loss. POIsam supports visualization-aware sampling accuracy loss function including 1 dimension and geospatial data. In the experiments, we use POIsam’s default theoretical error bound (5%) and confidence level (10%). This means that the sample produced by POIsam for every online query can have 5% or more error than Sample on the fly, at 10% chance. (4) SnappyData [3]: It applies data-system queries on the stratified samples, then returns an AVG of the query result. We use the cubed attributes as Query Column Set (QCS) in the experiments. Two versions of SnappyData are tested: 100MB and 1GB size pre-built samples. (5) Tabula: this is the system proposed in this paper. (6) Tabula*: this is Tabula but does not have the sample selection technique. (7) Sampling cube (FullSamCube): this approach creates a fully materialized data cube which holds a local sample for every cell. (8) Partially materialized sampling cube (PartSamCube): this approach directly executes the initialization query as shown in Section II. It does not use the initialization algorithm in Section III-B and sample selection in Section IV.

Evaluation metrics. We use the following metrics to measure the performance of each approach: (1) Initialization time: The time used to initialize the systems. We show the initialization time of Tabula, FullSamCube, PartSamCube, and SnappyData. (2) Memory footprint: The physical memory occupied by the pre-built / materialized samples in different approaches. SampleOnTheFly and POIsam do not incur extra memory space because they always draw samples on the fly. (3) Data-to-visualization time: it consists of (a) data-system: exe-

cuting data-system queries and running online sampling (only for SamFly and POIsam). (b) sample visualization: performing visual analysis tasks (exclude SnappyData). SnappyData has no visual analysis time because it takes a query and directly renders a conclusion, which is AVG. (4) Actual accuracy loss: the actual accuracy loss of the returned sample, calculated by the user-defined loss function. (5) Query answer size: the number of tuples sent to /processed by the dashboard.

Dataset and query attributes. We use the New York City taxi trips real dataset (NYCtaxi) [13] which is mentioned in the running example. We pre-cache the entire dataset into the cluster’s memory before initializing or using any approach. There are 7 categorical attributes used in the experiments: vendor name, pickup weekday, passenger count, payment type, rate code, store and forward, dropoff weekday. We use the first 4, 5, 6, 7 attributes in the predicates of data-system queries. Full data cubes built upon these attributes have 3 thousand, 17 thousand, 47 thousand and 151 thousand cells, respectively. The first 5 attributes are used by default.

User defined accuracy loss functions. (1) Statistical mean loss: this is Function 1 which checks against fare amount attribute of NYCtaxi data. (2) Geospatial heatmap-aware loss function: this is Function 2 (3) Linear regression loss: this is Function 3 (4) Histogram-aware loss: this is Function 2 but it is calculated on 1-dimension data (using Euclidean distance). The corresponding analysis task is shown in Figure 1. This function checks against NYCtaxi fare amount attribute so the distance unit is US dollar.

Analytics workload. We build a full data cube on n attributes then randomly pick 100 SQL queries (cells) from the cube. All compared approaches will then run these queries. Returned query answers are passed to the visualization dashboard in files. To quantitatively measure the visual analysis performance, we use two well-known analysis tools to record the corresponding visual analysis time: (1) Matlab: a renowned scientific computing software. We leverage it to draw histogram and geospatial heatmaps on results returned in corresponding accuracy loss based experiments. (2) Scikit Learn: a widely used machine learning Python library. We use it to calculate statistical means and linear regression functions of returned query answers. All analysis tasks are executed on the master machine of the cluster.

Cluster settings. All compared approaches are implemented with Apache Spark. We conduct the experiments on a cluster which has one master node and four worker nodes. Each machine has an Intel Xeon E5-2687WV4 CPU (12 cores, 3.0 GHz per core), 100 GB memory, and 4 TB HDD. We also install Apache Hadoop 2.6, Apache Spark 2.3, and SnappyData Enterprise 1.0.2.1 (column store).

A. Initialization time

In this section, we study the initialization time of different approaches (see Figure 8 and 10a). We vary the value of the user specified loss threshold θ . SampleFirst’s initialization time is omitted because the random sampling time is negligible compared to other approaches. We compare Tabula

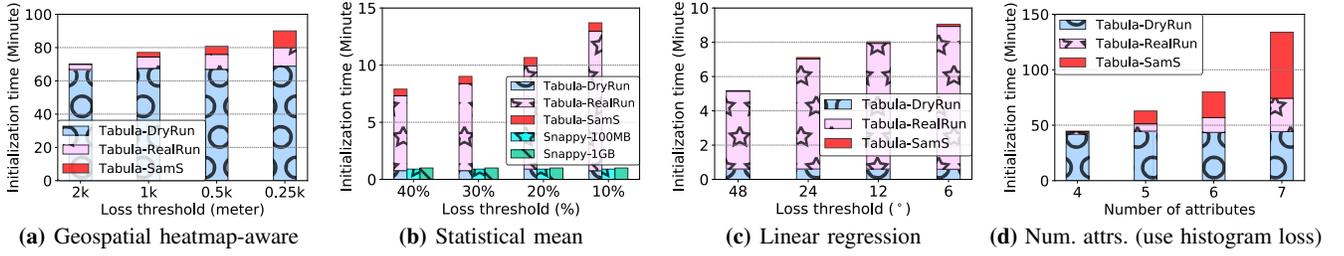


Fig. 8: Initialization time of different loss functions and different cubed attributes

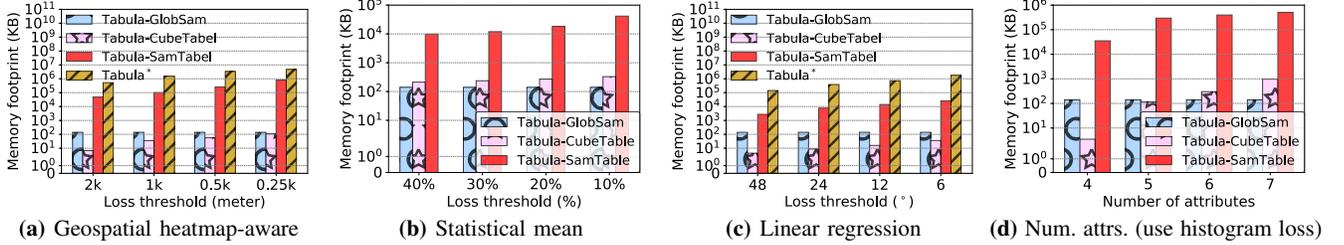


Fig. 9: Memory footprint of different loss functions and different attributes, in logarithm scale

against FullSamCube and PartSamCube on a small dataset, 5GB NYctaxi (see Figure 10a) using histogram-aware loss function, because FullSamCube and PartSamCube incur high initialization time and cannot scale to the full NYctaxi dataset. We also show the execution time of the dry run stage, real run stage and sample selection (denoted as SamS) of Tabula.

As shown in Figure 10a, Tabula takes around 40 times less initialization time compared to FullSamCube and PartSamCube. This makes sense because Tabula utilizes the dry run stage to skip many unnecessary GroupBys while other approaches run $2^n - 1$ GroupBy operations (n is the number of attributes). As depicted in Figure 8, the dry run stage execution time remains the same for different user specified loss thresholds but the overall initialization time of Tabula increases with the decrease of loss threshold. This is because a lower value of θ introduces more iceberg cells. Tabula always spends the same amount of time on the dry run stage to identify iceberg cells. However, if there are more iceberg cells, Tabula will take more time to draw local samples for iceberg cells in the real run stage and select representative samples in sample selection. It is also worth noting that geospatial heatmap-aware loss functions lead to Tabula consuming more time on the dry run stage while statistical mean costs the least time on that. This makes sense because the visualization-aware loss function involves complex tuple-to-tuple calculation compared to the statistical mean loss function.

B. Memory footprint

In this section, we study the memory footprint of Tabula for different accuracy loss functions. Tabula consists of three physical components in memory, global sample, cube table (Figure 4a) and sample table (Figure 4b). Tabula* does not have the sample table. As depicted in Figure 9, decreasing the value of θ leads to more memory space occupied by Tabula.

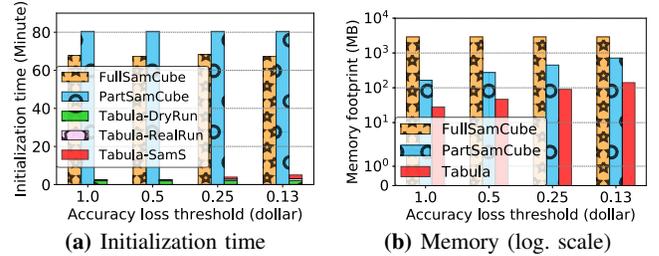


Fig. 10: Cubing overhead on 5GB NYctaxi data

That happens because a smaller θ results in more iceberg cells and more materialized local samples. Among the three components of Tabula, the global sample size remains the same for different θ values because this size is only related to the raw dataset scale according to Section III-B1. Both the cube and sample tables increase for smaller thresholds but the sample tables are at least 100 times larger than the cube tables. This makes sense because the cube table only contains simple iceberg cell information without any materialized samples. Tabula* is around 50 times larger than Tabula because it does not employ the sample selection technique.

As depicted in Figure 10b, the size of FullSamCube remains the same for different thresholds because it always materializes local samples for all cube cells regardless of thresholds while PartSamCube only materializes samples for iceberg cells. FullSamCube is around 50-100 times larger than Tabula while PartSamCube is around 5 - 8 times larger than Tabula because PartSamCube does not contain the sample selection technique.

C. Data-to-visualization time

We study the effect of various accuracy loss functions and threshold values on the total data-to-visualization time.

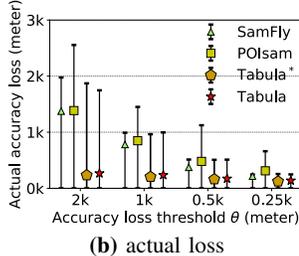
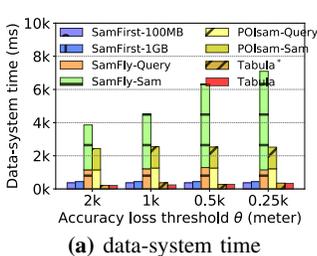


Fig. 11: Performance of geospatial heatmap-aware loss (unit: meter), 0.25 kilo meter \approx 0.004 (normalized distance)

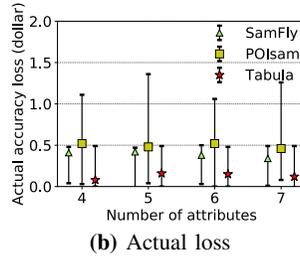
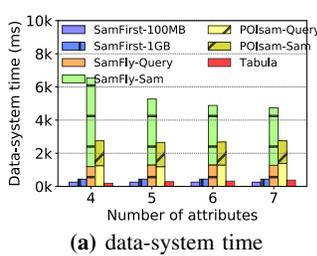


Fig. 12: Performance of different numbers of attributes in data-system queries, with histogram-aware loss function

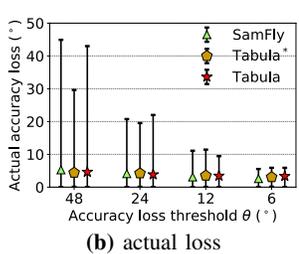
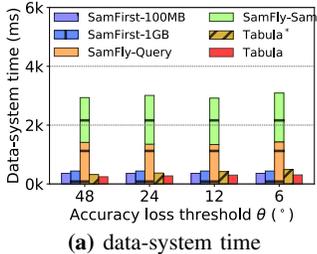


Fig. 13: Performance of linear regression loss (loss unit: degree $^{\circ}$)

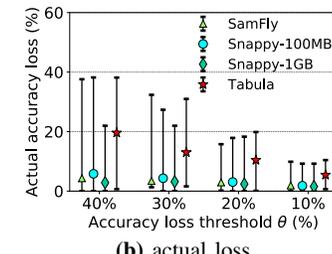
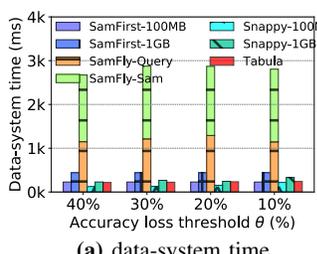


Fig. 14: Performance of statistical mean loss (loss unit: percentage)

We only run analysis tasks using the smallest accuracy loss thresholds (θ) for all accuracy loss functions and report the time in Table II.

In terms of data-system time, as shown in Figures 11a, 14a and 13a, the data-system time of SamFirst remains the same for all threshold values and loss functions because the only factor that can affect SamFirst is the size of its pre-built sample. POIsam and SampleOnTheFly are 10 times and 20 times slower than Tabula. This is because Tabula quickly returns either the materialized global or a local sample while POIsam and SampleOnTheFly always query the entire dataset and draw samples on the fly. Although we pre-cached the entire dataset and ran queries in parallel, the query time is still significantly large. POIsam can reduce the online sampling time, but its data-system time is still non-negligible.

TABLE II: Sample visualization time of different approaches

Approach	Geospatial heat map	Statistical mean	Regression
SamFirst-100MB	29 ms	0.02 ms	0.07 ms
SamFirst-1GB	59 ms	0.13 ms	0.15 ms
SamFly	146 ms	0.01 ms	0.29 ms
POIsam	143 ms	-	-
Tabula	390 ms	0.13 ms	1.33 ms
No sampling	330 sec	1.8 sec	1.9 sec

In terms of the sample visualization time, as shown in Table II, Tabula has the highest visual analysis time among all compared approaches because it sometimes returns the global sample (around 1000 tuples) for queries which hit non-iceberg cells while other approaches such as SampleOnTheFly and POIsam only return around 100 tuples for geospatial heat map loss function. However, analysis tools can still easily render results for Tabula within several hundred milliseconds because of the small size of global samples. Please note that it takes

around 3 orders of magnitude more time on analyzing the raw query result (without any sampling).

D. Studying the actual accuracy loss

In this section, we vary the accuracy loss threshold value (error bound in SnappyData) and evaluate the actual loss of samples returned by different approaches. The results are depicted in Figures 11b, 14b and 13b. Error bars indicate the minimum, average and maximum actual accuracy loss. We omit the actual accuracy loss of two SamFirst approaches in figures because their average accuracy loss is 20 times larger than other approaches for geospatial heatmap-aware loss functions and 4 times larger than others. As shown in the figures, as we decrease the threshold value, the actual loss of POIsam, SampleOnTheFly, SnappyData, and Tabula decreases. SampleOnTheFly, SnappyData and Tabula never exceed the thresholds. The actual accuracy loss of POIsam is around 1%-5% percent larger than SampleOnTheFly and sometimes, it exceeds the threshold. This makes sense because POIsam runs the greedy sampling function over a random sample. Tabula* has similar actual accuracy loss to Tabula because the sample selection technique does not necessarily increase accuracy loss. Also, SnappyData can guarantee the error-bound since the actual accuracy loss exceeds the threshold value, it accesses the raw table and runs queries and aggregation on-the-fly. Since SnappyData implements its own optimized block-based column store, its data-system time is still comparable to Tabula.

E. Impact of the number of attributes

In this section, we evaluate the impact of the number of attributes. We initialized Tabula on 4, 5, 6 and 7 attributes of NYCtaxi dataset and use these attributes in data-system

queries. Histogram-aware loss function with “0.5 dollar” threshold value is used in the experiment. 4 metrics are reported in Figure 8d, 9d and 12, respectively. The result of actual accuracy loss is omitted because the number of attributes has no effect on actual accuracy loss.

As depicted in the figures, in terms of initialization time, using more cubed attributes leads to higher execution time for all three initializing stages of Tabula because this introduces more cube cells as well as iceberg cells. Cube cells increase exponentially with more cubed attributes. But the number of attributes has relatively small impact on the dry run stage because the dominating part in this stage is building the first cuboid which requires a full table GroupBy. Other cuboids are derived from the first one.

In terms of memory footprint, the global sample size of Tabula remains the same because it is only related to the cardinality of the raw dataset. The sizes of the cube table and sample table increase with more cubed attributes. But the growing speed of the sample table becomes slower because, even though there are more and more iceberg cells and local samples, Tabula still can only materialize a small number of local samples as the representatives.

In terms of data-to-visualization time, using more attributes slightly increases the data-system time of Tabula because of larger cube tables and sample tables. SampleFirst approaches have the constant data-system time since they always perform a full sequential filtering on pre-built samples. The query time of SampleOnTheFly and POIsam remains the same for different numbers of attributes because they always perform a full sequential scan on the raw table. However, the visual analysis time of SampleFirst drops while using more attributes. This is because the queries will contain more predicates and lead to smaller query results. Similarly, the sampling time of SampleOnTheFly drops significantly while using more attributes. The online sampling time of POIsam does not change much because it first draws a random sample of the raw query result and the random sample size does not change much (controlled by the law of larger numbers [5]). The visual analysis time of Tabula slightly reduces while using more cubed attributes because Tabula returns materialized local sample for more queries in this situation.

VI. RELATED WORK

Data systems using pre-built samples. In the past two decades, several research works studied the implementation of classic sampling techniques such as random sampling, stratified sampling, cluster sampling, systematic sampling [14], and spatial sampling in database systems. However, samples pre-computed by classic sampling techniques may eventually lead to inaccurate results [15]. To enhance the accuracy of pre-built samples, recent systems [2], [1], [16], [17], [3] proposed sampling approaches that take into account different data populations. Sample+Seek [1] applies approximate query processing techniques on the data cube and offers a distribution precision guarantee. BlinkDB [2] and SnappyData [3] support approximate query processing with bounded error over

customized HIVE and Spark clusters. They create stratified samples over Query Column Set (QCS) to improve accuracy. But their pre-built stratified samples have no accuracy guarantee so the systems and only support classic OLAP aggregate measures such as SUM, COUNT, AVG.

Sample on the fly (i.e., Query time sampling). Approximate query processing [18], [19], [20] rely on sampling. Some approaches [21], [22] work on placing samplers inside join queries. Many approximate systems such as ABS [20] and Quickr [19] focus more on where to place the online sampler in the query plan. These approaches yield better accuracy and reduce data-system time but still inevitably access raw datasets on the fly. A recently proposed approach, namely Dice [23], [24], applies speculative query execution techniques to predict the human next query and prefetch the anticipated query answer upon a data cube [9] that holds pre-computed aggregate measures (e.g., SUM, COUNT, AVG). Such query speculation technique speeds up data-system over databases. However, Dice still runs an online sampler to return a sample for each query whereas Tabula directly fetches pre-built samples without accessing raw data because it exhausts all query results in advance. Moreover, Tabula provides deterministic accuracy loss guarantees while other approaches only guarantee bounded-error with a confidence level.

Data cube initialization algorithms. Gray et al. [9] proposed the concept of data cubes. Later, several papers [25], [26] proposed more advanced techniques to initialize data cubes with distributive and algebraic measures. These algorithms require the aggregate measures in the cube to be distributive or algebraic [9], [8] (1) Distributive: The measure of a cell can be computed solely based on the same measure of its descendant cells. For instance, SUM in Cell $\langle * : SUM \rangle$ is equal to the sum of $\langle Cash : SUM \rangle$, $\langle Credit : SUM \rangle$, $\langle Dispute : SUM \rangle$. (2) Algebraic: The measure of a cell can be computed based on several other types of measures in its descendant cells, e.g., AVG(). A distributive measure must be algebraic and an algebraic measure may not be distributive. All other measures are called holistic measures. These techniques focus on allocating cuboid groups to preserve data orders in the same group and avoiding unnecessary raw table accesses. Researchers [10] came up with the iceberg data cube and a Bottom-Up initialization algorithm to compute cube distributive measures with minimal overhead. Instead of persisting all measures, the iceberg cube by nature only stores a small number of aggregate measures. H-Cubing [27] and Star-Cubing [28] propose different iceberg cube initialization algorithms for algebraic measures. However, all aforementioned cubes only work for algebraic measures.

Methods that accelerate spatial visualization. Researchers proposed various approaches to accelerate spatial visualization process on big datasets. POIsam and VAS [5], [6] propose similar visualization-aware sampling approaches with accuracy loss guarantee. They shorten map visualization time but, in the case of spatial visualization dashboards, still perform sampling on the fly and have no optimization to reduce online data-system time. Wang et al. [16] propose a spatial indexing

mechanism that indexes spatial data by different levels such that their online sampler can run faster and produce more accurate results. But they do not provide a deterministic accuracy loss guarantee and cannot speed up queries that involve non-spatial attributes. Nano cube [4] and its variants [29] pre-materialize heat maps and other types of aggregates to answer online visualization requests. To mitigate the storage overhead, they design a set of complex visual encoding techniques to compress materialized aggregates. To query the cubes, a custom-made front-end visualization tool is required while Tabula is a middleware system which has no requirements on both visualization front-ends and underlying data systems. It is worth noting that the encoding techniques used in these cubes are complementary to Tabula such that they can work in concert with our system to further reduce the memory footprint. Moreover, none of the aforementioned approaches offer a generic system to uphold various user-defined visual analytics. Sampling cube'08 [30] has a similar name to the sampling cube maintained by Tabula but acts very differently; it builds an iceberg data cube on a sample of the raw table to speed up the cube initialization. The aggregate measures of this cube are simple algebraic measures such as AVG.

VII. CONCLUSIONS

In this paper, we presented Tabula as a middleware system to accelerate the spatial visualization dashboard. It can be easily extended, thanks to its generic user-defined accuracy loss function, to support various visual analytics. Tabula adopts a materialized sampling cube approach, which pre-materializes sampled answers for a set of potentially unforeseen queries. To achieve scalability, the system employs a partially materialized cube to only materialize local samples of iceberg cells based on the accuracy loss function and a sample selection technique to selectively materialize representative local samples. The system ensures the difference between the sample and the raw query answer never exceeds a user-specified accuracy loss threshold. According to the experiments, Tabula upholds different user-defined visual analysis: for complex analysis such as geospatial visual analytics and linear regression, it achieves up to 20 times less data-to-visualization time than SampleOnTheFly-like approaches; for OLAP analytics such as statistical mean (AVG), it exhibits similar performance to column-store based SnappyData. The proposed middleware system also occupies up to two orders of magnitude less memory footprint and an order of magnitude less initialization time than the fully materialized sampling cube approach. That makes Tabula a very practical and scalable approach to deploy in real geospatial data visualization dashboards.

VIII. ACKNOWLEDGEMENT

This work is supported by the National Science Foundation (NSF) under Grant 1845789.

REFERENCES

[1] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang, "Sample + seek: Approximating aggregates with distribution precision guarantee," in *SIGMOD*, 2016, pp. 679–694.

[2] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "Blinkdb: queries with bounded errors and bounded response times on very large data," in *The European Conference on Computer Systems, EuroSys*, 2013, pp. 29–42.

[3] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav, "Snappydata: A hybrid transactional analytical store built on spark," in *SIGMOD*, 2016, pp. 2153–2156.

[4] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *TVCG*, vol. 19, no. 12, pp. 2456–2465, 2013.

[5] T. Guo, K. Feng, G. Cong, and Z. Bao, "Efficient selection of geospatial data on maps for interactive and visualized exploration," in *SIGMOD*, 2018, pp. 567–582.

[6] Y. Park, M. J. Cafarella, and B. Mozafari, "Visualization-aware sampling for very large databases," in *ICDE*, 2016, pp. 755–766.

[7] D. A. Freedman, *Statistical models: theory and practice*. Cambridge University Press, 2009.

[8] A. Nandi, C. Yu, P. Bohannon, and R. Ramakrishnan, "Distributed cube materialization on holistic measures," in *ICDE*, 2011, pp. 183–194.

[9] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatarao, F. Fellow, and H. Pirahesh, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub totals," *Data Mining Knowledge Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[10] K. S. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *SIGMOD*, 1999, pp. 359–370.

[11] R. J. Serfling, "Probability inequalities for the sum in sampling without replacement," *The Annals of Statistics*, pp. 39–48, 1974.

[12] R. B. Allan and R. Laskar, "On domination and independent domination numbers of a graph," *Discrete Mathematics*, vol. 23, no. 2, pp. 73 – 76, 1978.

[13] "New york city taxi and limousine commission," www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[14] W. Hays, *Statistics*. College Publishing, 1981.

[15] S. Chaudhuri, B. Ding, and S. Kandula, "Approximate query processing: No silver bullet," in *SIGMOD*, 2017, pp. 511–519.

[16] L. Wang, R. Christensen, F. Li, and K. Yi, "Spatial online sampling and aggregation," *PVLDB*, vol. 9, no. 3, pp. 84–95, 2015.

[17] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *ACM SIGMOD Record*, vol. 29. ACM, 2000, pp. 487–498.

[18] S. Wu, B. C. Ooi, and K. Tan, "Continuous sampling for online aggregation over multiple queries," in *SIGMOD*, 2010, pp. 651–662.

[19] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *SIGMOD*, 2016, pp. 631–646.

[20] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo, "ABS: a system for scalable approximate queries with accuracy guarantees," in *SIGMOD*, 2014, pp. 1067–1070.

[21] F. Li, B. Wu, K. Yi, and Z. Zhao, "Wander join: Online aggregation via random walks," in *SIGMOD*, 2016, pp. 615–629.

[22] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *SIGMOD*, 1999, pp. 287–298.

[23] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi, "Distributed and interactive cube exploration," in *ICDE*, 2014, pp. 472–483.

[24] P. Jayachandran, K. Tunga, N. Kamat, and A. Nandi, "Combining user interaction, speculative query execution and sampling in the DICE system," *PVLDB*, vol. 7, no. 13, pp. 1697–1700, 2014.

[25] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, and S. Sarawagi, "On the computation of multidimensional aggregates," in *VLDB*, 1996, pp. 506–521.

[26] V. Harinarayan, A. Rajaraman, and J. D. Ullman, "Implementing data cubes efficiently," in *SIGMOD*, 1996, pp. 205–216.

[27] J. Han, J. Pei, G. Dong, and K. Wang, "Efficient computation of iceberg cubes with complex measures," in *SIGMOD*, 2001, pp. 1–12.

[28] D. Xin, J. Han, X. Li, and B. W. Wah, "Star-cubing: Computing iceberg cubes by top-down and bottom-up integration," in *VLDB*, 2003, pp. 476–487.

[29] C. A. de Lara Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba, "Hashedcubes: Simple, low memory, real-time visual exploration of big data," *TVCG*, vol. 23, no. 1, pp. 671–680, 2017.

[30] X. Li, J. Han, Z. Yin, J. Lee, and Y. Sun, "Sampling cube: a framework for statistical olap over sampling data," in *SIGMOD*, 2008, pp. 779–790.