QPV: An Input Control Component For Progressive Visualization Analytics [Work-in-progress]

Xin Zhang
University of California, Riverside
Riverside, USA
xzhan261@ucr.edu

Ahmed Eldawy
University of California, Riverside
Riverside, USA
eldawy@ucr.edu

ABSTRACT

Progressive visual analytics enable data scientists to efficiently explore large datasets and examine progressive results with low latency. Most progressive visualization frameworks use a progressive query processing module that controls the quality of the results and then feeds these results into a visualization module. The goal is to avoid poor-quality progressive results which could mislead data scientists. This method misses some optimization opportunities as it improves the quality of the intermediate result while ignoring how this result affects the final visualization. This work presents a work-in-progress quality-aware progressive visualization input control component, named *QPV*. The key idea of the proposed framework is to integrate the visualization module into the progressive query results so that the quality control takes into account the final visualization. With limited computational resources, *QPV* solves an optimization problem to allocate resources and alleviate the misleading effects in the progressive plots.

VLDB Workshop Reference Format:
Xin Zhang and Ahmed Eldawy. QPV: An Input Control Component For Progressive Visualization Analytics [Work-in-progress]. VLDB 2024 Workshop: Relational Models.

VLDB Workshop Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xin-aurora/QualityProgressiveJoin.git.

1 INTRODUCTION

Progressive data processing is a popular tool for data scientists to explore large datasets [2, 4, 9, 17, 18]. It splits large datasets into multiple small batches and progressively processes each data batch. Each progressive round finishes quickly to keep users engaged and active. Users often examine the progressive results by visualizing them. Without further processing, the progressive results can be visualized into scatterplot [1] and line chart [16]. Choropleth map [14, 15], bar chart [11], pie chart [8, 17], trendline [13] and heatmap [3, 6, 13]. Users observe the progressive results to start further processing or make decisions on the currently running query. Poor-quality progressive results might negatively impact further analyses and mislead data scientists, leading to cognitive biases [12]. Therefore, producing progressive results with good quality is the most important task in progressive processing frameworks.

Most frameworks define result quality as the similarity between the progressive results and the complete results [1–4, 13, 17]. We divide existing frameworks into three categories based on quality control strategies. Frameworks in the first category [2–4, 9] optimize the progressive input before query processing based on pre-defined input computation goals, such as the data distribution of the progressive inputs or preference score function. Frameworks in the second category [1, 13, 18] optimize the progressive results during query processing until the results satisfy the desired quality bound, such as error bound or specific sampling condition. A recently proposed progressive join framework, named *QPJ* [17], considers both progressive input and output and belongs to the third category. It batches and partitions the progressive input following the same strategies as the first categorized frameworks. The output control of *QPJ* temporarily hides some results in memory from the current round to keep the output progressive results having a similar result distribution to the estimated complete result.

In progressive visualization, result quality always links to the values used to produce the plot. Existing frameworks aim to compute *good* estimated aggregated values to the complete results and evaluate the accuracy of estimations [3, 5, 6, 11, 13]. The progressive results are considered as samples of the complete results. Statistical methods, such as confidence intervals, are commonly used metrics to evaluate the accuracy of progressive results. However, when the aggregation query includes a GroupBy clause and the user is looking for rankings among the groups, the user might still be confused. Because the confidence intervals of different groups might overlap, the user cannot determine the exact rankings.

Consider a sociologist who wants to analyze the usage of social media in the United States. The sociologist applies a progressive spatial join query on the Tweets dataset and US-States dataset and visualizes the results in the choropleth map shown in Figure 1. The Tweets dataset contains a set of tweet objects and each object has a spatial point attribute as the posted location. The US-States dataset consists of polygons and each polygon represents the geographical boundaries of a state. The progressive results are choropleth maps with five classes based on the number of tweets in each state. After 10 seconds, the system produces 10% of the results. The confidence intervals of several states in adjacent classes overlap with each other so that the system cannot assign them to a single class. Therefore, the system visualizes them with unsure colors. In this example, we apply two different input control strategies. The downside results are computed based on a regular progressive input builder and the upside results are computed based on our proposed *QPV*. *QPV* analyzes the misleading information in the visualizations and

Proceedings of the VLDB Endowment. ISSN 2150-8097.

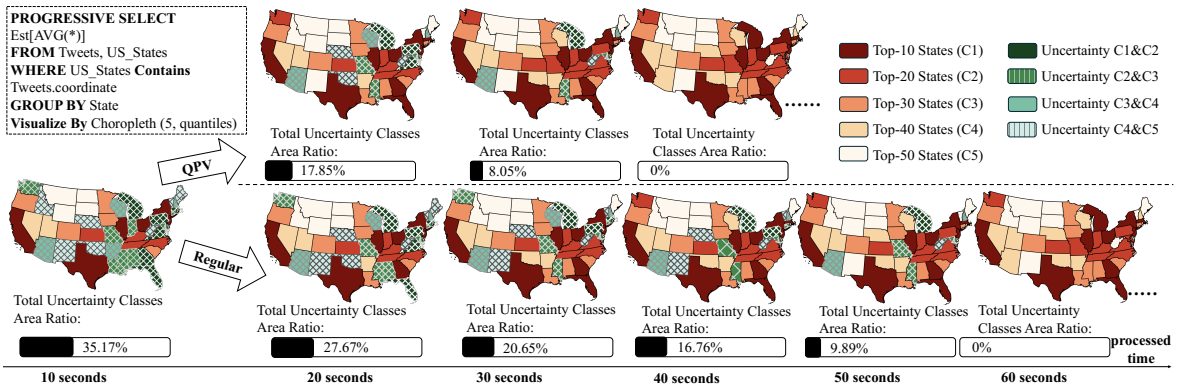

Figure 1: Progressively joining Tweets dataset with US-State dataset. The downside progressive results are computed from the regular progressive input control method. The upper progressive results are computed from our proposed method *QPV*. Note: The joined US-State dataset contains Washington, D.C. and 48 states.

computes a better progressive resource allocation plan. Intuitively, *QPV* controls the system to process more data in the unsure classes area. As we can see in Figure 1, the area of polygons with unsure classes is reduced sooner in the *QPV*'s progressive results than the baseline method's progressive results.

To alleviate the misleading effects, we can try to shrink the confidence intervals that overlap with others in the next round. A sampling algorithm [7], RapidSampling, also considers a similar approach to shrink the confidence intervals and make the trends of the aggregation values clearer. However, this algorithm is designed for aggregation in a single dataset and cannot be applied to input control for multiple datasets and progressive processing control.

In this work, we propose a quality-aware progressive visualization input control component named *QPV*, which allocates computation resources for the next progressive computation round based on visualization results of the current round. *QPV* can be integrated into the existing framework to replace its input control component. We consider the groups with uncertainty visualization in the plots as the target groups. For example, bars with overlapping heights in the bar chart and the classes with unsure colors in the choropleth map. After each progressive join computation, *QPV* collects the statistics of the inputs and outputs and estimates the input data sizes for the target groups. With bounded computation resources, *QPV* solves an optimization problem to allocate the resources wisely so that the misleading effects can be reduced in the next round.

2 PROGRESSIVE JOIN VISUALIZATION FRAMEWORK OVERVIEW

Considering big data scenarios, most progressive frameworks [2, 4, 17] are designed for distributed settings. Figure 2 shows two types of distributed progressive join visualization frameworks. The first adopts the quality-aware progressive visualization *QPV* as the progressive input builder (Figure 2 (a)). The second uses the regular progressive input builder (Figure 2 (b)). In this section, we introduce how these two types of progressive input builders work.

Framework overview. We summarize four common components from the existing frameworks, which are progressive input builder in orange and black, partitioners in blue, processors in green, and progressive output builder represented as a visualization

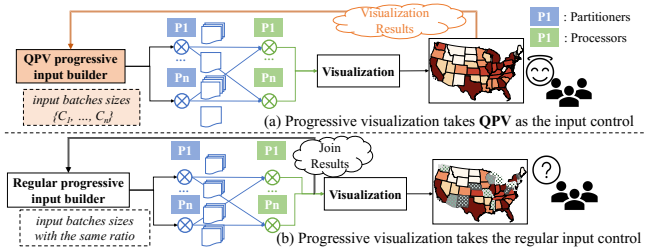

Figure 2: Different progressive join visualization frameworks which adopt the *QPV* as progressive input builder (a) and the regular progressive input builder (b).

box. Given the input datasets, join query, and parameters such as the number of progressive rounds and partitions, partitioners load input datasets and send partitioned inputs to processors. The processors process the join query and send progressive results to the progressive output result builder. In visualization frameworks, the results are plots drawn from progressive join results. The progressive output result builder constructs progressive plots and returns them to the user.

The regular input control component. Regular progressive input builders decide the size of progressive input before the query processing based on batching strategies such as equal-size batching. Frameworks partition the input data based on the join key and make sure the progressive inputs have a similar data distribution to the whole dataset. In Figure 2 (b), the size ratio of the input batch size for each partition is the same. When processors finish the computation and the plot is returned to the user, the framework starts the next progressive round. To evaluate the quality of the progressive results, we can compute the confidence intervals based on different statistical methods [3, 5, 6, 11, 13].

The quality-aware progressive visualization *QPV*. Based on plot type and join query, we extract the target groups with overlapping confidence intervals and contain the misleading information in the progressive plots. We summarize two types of misleading information. The first type relates to the ranking in the plot, such as assigning the colors in choropleth maps and heatmaps. The second type relates to the ranking in the input join query, such as plotting top-k. *QPV* takes the input and output statistics of the target groups

to compute input batch sizes for all partitions in the next round. Different from the regular input control component, *QPV* combines the information from the plot of the current round to allocate the computation resources for the next progressive round.

3 PROGRESSIVE INPUT BUILDER

QPV solves an optimization problem to allocate the computation resources for the next progressive round. The resources refer to the number of input items in each partition. The goal is to allocate more resources to compute more results from the uncertainty areas.

3.1 Confidence Interval Computation

Each progressive round produces part of the complete results and the progressive results can be viewed as samples of the complete results. In each round, we compute the confidence intervals of the aggregated values as the quality measurement and also draw the confidence intervals in the plots returned to users. We apply Hoeffding's Inequality to compute the confidence intervals.

Estimated Aggregation Results. In the following, we introduce the estimation analysis for joining two datasets. Assume we join dataset S with dataset R, group the results based on GroupBy keys, and return the estimated aggregation value of each group in the complete join results. We also assume the GroupBy keys are only in dataset S and the number of GroupBy keys is q. Then, there will be q aggregation results $\{c_1, ..., c_q\}$ in each round. Based on the ratio of the total input size to the processed input size, we estimate the aggregation values of the complete results $\{C_1, ..., C_q\}$:

$$C_j = c_j \cdot \big(|S_j| \cdot |R|\big) / \big(|S_{n_j}| \cdot |R_n|\big), \tag{1}$$

The $|R_n|$ is the processed input size up to the current round in dataset R, $|S_{n_j}|$ is the processed input size of group j to the current round in dataset S, $|R|$ is the total input size in dataset R, and $|S_j|$ is the total input size of group j in dataset S. Since dataset S contains the GroupBy key and dataset R does not have, therefore, the estimation statistics of the two datasets are slightly different. The estimation method in Equation 1 is extended from the ripple joins [5] to include the GroupBy clause.

Confidence interval computation. We apply Hoeffding's bound, which is a distribution-free bound, to compute the confidence interval of the estimations. The Hoeffding's theorem states that with probability at least $1 - \varepsilon$,

$$\big|C_j - E[C_j]\big| \leq (b_j - a_j)\sqrt{(\log(2/\varepsilon))/2n} = t, \tag{2}$$

where ε is the error bound of estimations, C_j is the estimated average value computed based on Equation 1, $E[C_j]$ is the expected value of the average value which is the average value of the complete results, a_j and b_j is the lower and upper bound of the GroupBy key j's value, and n is the sample size which is the join result size in group j. For COUNT aggregation, $b - a$ is 1. For SUM and AVG aggregation, the values of a and b should consider the value bound of the GroupBy key j. Let t represent the right side of Equation 2, the confidence interval (CI) of the estimated aggregation value is $[C_j - t, C_j + t]$ and the length of CI is $2t$.

3.2 Progressive Input Size Computation

In this subsection, we introduce definitions to define the computation resources allocation problem.

Assume there are h join keys $\mathcal{K}$=$\{k_1, ..., k_h\}$ in the two joined datasets S and R and n processors $\mathcal{P}$=$\{P_1, ..., P_n\}$ in the system, we divide the join keys $\mathcal{K}$ into n disjoint sets and let each processor handle one partitioned subset from each dataset.

Given an integer d as the size of the processing unit, we split each S_i into $|S_i|/d$ equal-size processing units. We further divide the $|S_i|/d$ processing units into y input batches, where y is the number of progressive rounds. For example, a partition contains 100 items, the size of the processing unit is 5, and the number of progressive rounds is 10. Each partition contains 100/5=20 processing units and each input batch contains 20/10=2 processing units. We consider the computation resources as the number of processing units B in each progressive round. The computation resources allocation problem is to decide how many input batches from each partition to process in the next round.

Assume there are q GroupBy keys $\mathcal{G}$=$\{g_1, ..., g_q\}$. The progressive plots are constructed based on the estimated aggregation values $\{C_1, ..., C_q\}$ computed by Equation 1. GroupBy keys are visualization units, such as the bars in bar charts, the polygons in choropleth maps, and the pixels in heatmaps. Although the join keys in one partition are disjoint with the join keys in other partitions, the GroupBy keys of different partitions are not disjoint.

For each estimated aggregation value C_j, we compute its confidence interval by Equation 2. The choropleth maps and heatmaps rely on the ranking of the aggregation values to assign the color. However, if the groups have overlapping confidence intervals, we do not know how to rank them. We refer to them as target groups and their visualizations as uncertainty areas in the plots.

Assume there are m target groups $\mathcal{T} \subseteq \mathcal{G}$, where $\mathcal{T}$=$\{g_1, ..., g_m\}$. We first define the uncertainty of each group and compute the loading factor, which represents the necessity of a group to return more results in the next batch. The uncertainty of a group j comes from two aspects: the length of the confidence interval of the estimated aggregated value (computed by Equation 2) and the size of visualization unit Vis_j in the plot. The area of polygons and pixels is the visualization unit for choropleth maps and heatmaps. The height of bar is the size of the visualization unit for bar charts.

Definition 1 (Uncertainty un_j and Loading Factor lf_j). *We define the uncertainty un_j and loading factor lf_i as follows:*

$$un_j = t_j \cdot Vis_j, \text{ and } lf_j = \frac{un_j}{\sum_{j=1}^{t} un_j} \tag{3}$$

Definition 2 (Result Rate $RR_{i,j}$). *For partition i, given aggregation values of the target groups $\{C_{i,1}, ..., C_{i,m}\}$ and $\{|S_{n_{i,1}}|, ..., |S_{n_{i,m}}|\}$ and $|R_{n,i}|$ as the amount of data has been processed, the result rate of group j in partition i is $RR_{i,j} = C_{i,j} / \big(|S_{n_{i,j}}| \cdot |R_{n,i}|\big)$.*

Definition 3 (Loading Score LS_i). *Given loading factors $\{lf_1, ..., lf_m\}$ and result rates $\{RR_{i,1}, ..., RR_{i,m}\}$, the loading score of partition i is $LS_i = \sum_{j=1}^{m} \big(lf_j \cdot RR_{i,j}\big)$.*

Given a set of result tuples with the same join key k, GroupBy keys of different result tuples might be different. If a result tuple has join key k and GroupBy key g, we say k is linked to g.

Definition 4 (Diversity Score Div_i). *The diversity score of the join keys in partition i for group j is:*

$$Div_{i,j} = \frac{\textit{the number of join keys linked to group } j}{\textit{the total number of join keys}}. \tag{4}$$

Given loading factors $\{lf_1, ..., lf_m\}$ *and diversity score* $\{Div_{i,1}, ..., Div_{i,m}\}$, *the diversity score of partition* i *is* $Div_i = \sum_{j=1}^{m} (lf_j \cdot Div_{i,j})$.

Given the progressive results computed from partition i and the target groups $\mathcal{T}$, we compute the score s_i as its "contribution" to return more results in $\mathcal{T}$. s_i consists of loading score LS_i and diversity score Div_i, where $s_i = \lambda \cdot LS_i + (1-\lambda) \cdot Div_i$. The λ $(0 \leq \lambda \leq 1)$ is a weight factor to tune importance between LS_i and Div_i. Next, we formally define the computation resource allocation problem.

Definition 5 (Computation Resources Allocation Problem). *Given* n *partitions* $\mathcal{P} = \{P_1, ..., P_n\}$, *progressive visualization results,* m *target groups* $\mathcal{T} = \{g_1, ..., g_m\}$, *importance factor* λ, *the number of available input batches* y, *and the computation resources* B, *the goal of* QPV *is to find the optimal solution to allocate* B *processing units to* $\mathcal{P}$ *which maximizes the overall score of all the partitions:*

$$\max \quad \sum_{i=1}^{n} s_i x_i$$

$$\text{s.t.} \quad \sum_{i=1}^{n} x_i \leq B \text{ and } x_i \in \{0, 1..., y\},$$

$$s_i = \lambda \cdot LS_i + (1 - \lambda) \cdot Div_i$$

The output of the computation resources allocation problem is n integer numbers $\{x_1, ..., x_n\}$, where each x_i represents the number of processing units of partition i in the next progressive round.

3.3 Preliminary Experimental Results

We designed two algorithms for *QPV*. The first algorithm detects the target groups. It ranks the estimation objects in descending order, where each estimation object consists of an estimation and confidence intervals. For each object in the class boundary, the algorithm finds the objects whose confidence interval overlaps with the confidence interval of the boundary object. The overlapping objects are in the target groups.

The second algorithm solves the resource allocation problem. The algorithm sorts the partitions in descending order based on the partition score and greedily selects the partitions with the largest score until reaching the budget B. To avoid loading inputs only from a few partitions, we also set an empirical limit on the maximum number of batches to process in each progressive round.

Figure 3 shows a set of preliminary experiment results. In this experiment, we use *QPV* and the regular input control component to process a progressive equi-join query on the samples of eBird dataset [10] and the USCities dataset. The eBird dataset contains 5 million records and the USCities dataset contains information about cities and states from 51 U.S. states. The number of progressive rounds is 20. The progressive results are choropleth maps. We plot the ratio of the uncertainty area in each progressive map (similar to the example in Figure 1). The x-axis shows the progressive rate and the y-axis shows the ratio of uncertainty area. In Figure 3, we demonstrate that the ratio of the uncertainty area in the map is smaller by applying *QPV*.

3.4 Future Works

QPV is work-in-progress. We will expend it in the following aspects: **1. Input Parameters:** We will support other types of plots that rely on aggregation queries, such as bar charts. The current visualization groups are decided based on GroupBy keys. In addition

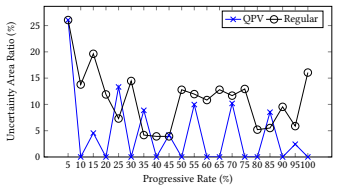

Figure 3: Compare the total uncertainty area ratio (%) in choropleth maps computed by *QPV* and baseline method.

to one group key, we will also consider multi-attribute clustering and algorithm-based clustering. **2. Problem Settings:** The current resource allocation problem assumes the join partitions and aggregation groups are fixed. We will also consider dynamic updates to the partitions so that the workloads of all partitions are balanced. **3. Solutions to the Problem:** The solution to the current problem is a simple greedy algorithm based on the partition score. We will refine the solution to incorporate the new problem settings. We will also consider other statistical methods to compute the confidence intervals of the estimated aggregation values. The Hoeffding's bound is quite wide when the data range is wide. A wide confidence interval might overlap with multiple groups so that multiple groups will be added to the target groups. **4. Evaluations:** We will verify *QPV* with real-world datasets and join queries. In addition, we are looking for other input control methods to compare with *QPV*.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation (NSF) under grants IIS-1838222, CNS-1924694, IIS-2046236, and IIS-1954644.

REFERENCES

[1] Liming Dong et al. 2020. Marviq: Quality-Aware Geospatial Visualization of Range-Selection Queries Using Materialization. In *SIGMOD*. 67–82.
[2] Chandramouli Badrish et al. 2013. Scalable progressive analytics on big data in the cloud. *PVLDB* 6, 14 (2013), 1726–1737.
[3] Moritz Dominik et al. 2017. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*. 2904–2915.
[4] Wee Hyong Tok et al. 2008. A stratified approach to progressive approximate joins. In *EDBT*. 582–593.
[5] Peter J Haas and Joseph M Hellerstein. 1999. Ripple joins for online aggregation. *SIGMOD* 28, 2 (1999), 287–298.
[6] Jaemin Jo et al. 2019. Proreveal: Progressive visual analytics with safeguards. *TVCG* 27, 7 (2019), 3109–3122.
[7] Albert Kim et al. 2015. Rapid Sampling for Visualizations with Ordering Guarantees. *PVLDB* 8, 5 (2015), 521–532.
[8] Yuyu Luo et al. 2020. Visclean: Interactive cleaning for progressive visualization. *PVLDB* 13, 12 (2020), 2821–2824.
[9] Procopio Marianne et al. 2019. Selective wander join: Fast progressive visualizations for data joins. In *Informatics*, Vol. 6. MDPI, 14.
[10] Cornell Lab of Ornithology. 2020. eBird Basic Dataset. https://star.cs.ucr.edu/?eBird
[11] Ameya Patil et al. 2023. Studying Early Decision Making with Progressive Bar Charts. *TVCG* 29, 1 (2023), 407–417.
[12] Marianne Procopio et al. 2021. Impact of cognitive biases on progressive visualization. *TVCG* 28, 9 (2021), 3093–3112.
[13] Rahman Sajjadur et al. 2017. I've seen" enough" incrementally improving visualizations to support rapid decision making. *PVLDB* 10, 11 (2017), 1262–1273.
[14] Zhuocheng Shang and Ahmed Eldawy. 2023. Viper: Interactive Exploration of Large Satellite Data. In *SSTD*. 141–150.
[15] Alex Ulmer et al. 2023. A Survey on Progressive Visualization. *TVCG* (2023).
[16] Yunhai Wang et al. 2023. OM3: An Ordered Multi-level Min-Max Representation for Interactive Progressive Visualization of Time Series. *SIGMOD* 1, 2 (2023), 1–24.
[17] Xin Zhang and Ahmed Eldawy. 2023. Less is More: How Fewer Results Improve Progressive Join Query Processing. In *SSDBM*. 1–12.
[18] Zhao Zhuoyue et al. 2020. Efficient join synopsis maintenance for data warehouse. In *SIGMOD*. 2027–2042.