QPJVis Demo: Quality-boost Progressive Join Query Processing System

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

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

ABSTRACT

Progressive query processing enables data scientists to efficiently analyze and explore large datasets. Data scientists can start further analyses earlier if the progressive result can represent the complete results well. Most progressive processing frameworks carefully control which parts of the input to process in order to improve the quality of progressive results. The input control strategies work well when the data are processed uniformly. However, the progressive results will be biased towards the join keys if the processed data are not uniform. A recently proposed input&output framework named QPJ corrects the bias by temporarily hiding some results. The framework dynamically estimates the distribution of the complete result and outputs progressive results with a similar distribution to the estimated complete result. This demo presents QPJVis, which is a progressive query processing system designed to inherently process the progressive queries using the QPJ framework. Additionally, we also implement an input control framework, Prism, in QPJVis so that users can compare the difference between the input&output framework and a purely input framework.

PVLDB Reference Format:
Xin Zhang and Ahmed Eldawy. QPJVis Demo: Quality-boost Progressive Join Query Processing System. PVLDB, 17(12): 4345 - 4348, 2024.
doi:10.14778/3685800.3685871

1 INTRODUCTION

Exploring large datasets is a time-consuming task. Due to the big data volume and computation complexity, it usually takes minutes or even hours to finish one query [12]. Researchers propose progressive query processing to efficiently explore big data. Progressive processing splits large datasets into small batches and processes each data batch progressively. Each progressive computation round takes a few seconds to keep the users engaged and active [15]. Users take the progressive answers to start the further processing early on without taking hours to wait for the entire computation to complete on the whole dataset. Progressive processing is a popular tool to explore and analyze large datasets on join [4, 5, 7–9, 11, 15, 16], aggregation [4, 7, 13], and visualization [3, 6, 13].

Traditional progressive processing frameworks adopt different input control techniques to ensure the quality of progressive results. We classify them into two categories. Frameworks in the first category optimize the progressive input before the query processing: these systems [4–8, 15] control the input that goes into query processing. They manipulate the progressive input based on pre-defined input computation goals. The goals can be the number of items in progressive input, data distribution of progressive input, and preference score function. However, these frameworks always return the progressive results directly without further optimizations, which can lead to misleading results. Poor quality progressive results can negatively impact further analyses and mislead data scientists to have cognitive biases [12].

Frameworks in the second category optimize results during query processing: a process ingests and processes more input until the output reaches a desired quality bound [3, 9, 13]. They manipulate the query processing based on result quality goals, for example, error bound or sample strategies. These frameworks take a longer time to process more data to reach the computation goal, which can compromise the advantage of quick response provided by progressive processing. Besides, they might provide approximate answers [3, 9] instead of exact answers.

We propose a new quality-aware progressive join framework recently [16], named QPJ, to address the limitations in existing frameworks. QPJ employs a flexible input&output control mechanism to adjust input and output individually in each progressive computation cycle. The input control follows existing single-choice control frameworks to batch and partition the progressive input. The output control maximizes the progressive output rate while preserving result quality through distribution similarity to the estimated complete result. QPJ temporarily hides some results in memory from the current round and releases them in the following rounds. Simply speaking, outputting less with better quality. QPJ uses a flexible two-direction weighted sampling strategy. It adopts the weighted sampling to add results into the output view when the size of the temporary hold result is large. On the other side, it will use reverse weighted sampling to filter out results from the output view when the size of the temporary hold result is small. Additionally, QPJ adopts a dynamic strategy to estimate complete result distribution. In this work, we demonstrate QPJVis which is a progressive processing system utilizing QPJ to process equi-join and spatial join queries. QPJVis also contains a web interface that allows the users to enter queries and the parameters and visualize the progressive results of the queries.

2 QPJVIS SYSTEM OVERVIEW

QPJVis contains two parts: (1) A web interface receives user-typed queries and parameters and visualizes the progressive results. Besides, QPJVis also allows users to store the query results in disk

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685871

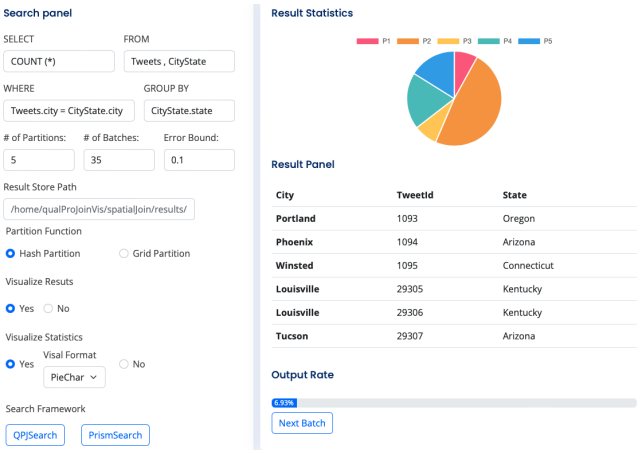

Figure 1: The screenshot of QPJVis web interface.

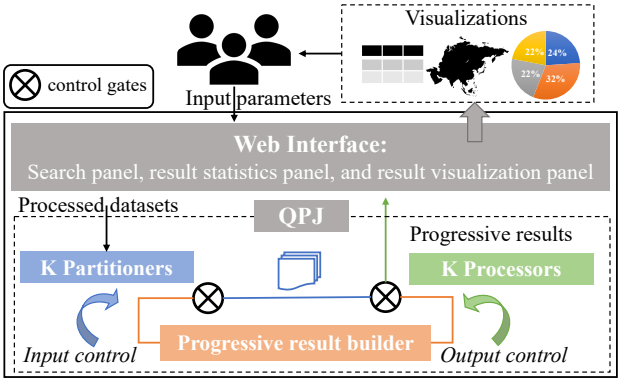

Figure 2: The system architecture of QPJVis.

files for further analysis. (2) A progressive query processing component QPJ that produces the quality preserved progressive results in multiple progressive rounds. The number of rounds is given by the user, we call the data of each round as batch.

2.1 QPJVis Web Interface

Figure 1 shows a screenshot of the web interface of QPJVis. It contains three panels: the search panel, the result statistics panel, and the result panel. In the search panel, users can enter the join query and tune query parameters. The query parameters include the partition function, the number of partitions, the number of progressive batches, the error bound, and the result file path. Additionally, we also allow users to enable and disable result statistics visualization, result visualization, and quality search. Users can click "QPJSearch" button to perform a quality-boost progressive search by QPJ framework or click "PrismSearch" button to perform a regular progressive search by Prism framework [4]. In the result statistics panel, we provide three ways to visualize the result statistics which are bar chart, pie chart, and table. In the result panel, we show the progressive output rate in a process bar and visualize the progressive results. The progressive spatial join results are shown in the map. The spatial data visualization is implemented by OpenLayers APIs [14]. The progressive equi-join results are listed in the table. To save space, we only show the samples of progressive equi-join results. Additionally, users can click "next batch" button to request the progressive results of the next batch.

2.2 Progressive Query Processing

QPJVis processes the progressive queries by a quality-aware progressive join framework QPJ [16]. QPJ consists of three components: partitioners, join processors, and a progressive results builder. They are drawn into different colors in Figure 2.

2.2.1 Partitioners and Join Processors. Assume user gives a join query that join dataset S and dataset R, the number of partitions k, and the number of progressive computation rounds s. In round i, QPJ provides the progressive answer up to batch i. QPJ computes the batch size of each round based on the number of progressive rounds and the size of each dataset. QPJ contains two functions to compute the batch size: equal-size split function and balance split function. The equal-size split function produces equal-sized batches. The balance split function ensures each round processes the equal size of candidate pairs, which is $(m_1 * m_2)/s$, where m_1 and m_2 are the sizes of the two datasets and s is the total computation round.

QPJ divides dataset S and dataset R into disjoint k partitions and collects the statistical information used for batching and result size estimation. The system assigns k processors (in green) to process the data from k partitions (in blue) and produces the progressive results in s rounds. QPJ adopts hashing partition for equi-join processing and grid partition for spatial join processing. The hashing partition [4, 8] is widely used to separate the non-spatial data based on the joined attribute. QPJ groups the relational data based on the joined attribute and puts them to a different partition. To process spatial data, QPJ takes grid partition [15]. It divides the input data space into equal-size grid cells and hashes each grid cell to a different partition.

QPJ applies hash join algorithm [10] to process the equi-join query and applies the Plane Sweep algorithm [2] to hand the spatial join query. Given the input batches, QPJ runs three join steps for each partition: (1) The new batch from dataset1 joins the new batch from dataset2; (2) The new batch from dataset1 joins the existing batches from dataset2; (3) The new batch from dataset2 joins the existing batches from dataset1. In each round, QPJ computes the query results up to the current batch.

2.2.2 Progressive Results Builder. QPJ controls the input and output of the progressive computation through the progressive results builder. In Figure 2, circle black symbols represent the input control and output control gates. In each round, the partitioners (in blue) and processors (in green) send the statistical information through input control gates to the progressive results builder (in orange). The statistical information includes input batch size and progressive results size. The progressive results builder estimates the complete result size and computes the output progressive result size of each partition for the current round. Progressive results builder sends progressive output result sizes of all partitions and their statistical information through the output control gates to processors. Processors release the progressive results based on the output results

size. The rest of the results are buffered in memory temporarily and will be released together with the progressive results of the following rounds. Compared with purely input control frameworks, QPJ outputs a few results so that the partitioned data distribution of the outputted progressive results is closer to the partitioned data distribution of complete results.

Ground Truth Estimation. To better estimate the ground truth, progressive results builder adopts multiple estimation methods and combines them dynamically by different importance factors. The estimated ground truth result size $E_{dynamic}$ as follows:

$$n\hat{GT}_{i,j} = E_{dynamic} = \frac{i}{s}E_{join_i} + \frac{s-i}{s}E_{selectivity}, \quad (1)$$

where i/s and $(s-i)/s$ are importance factors. s is the total number of progressive rounds and i represents the current round. E_{join_i} is the sampling estimation of round i and $E_{selectivity}$ is the estimation computed by applying selectivity estimation. The sampling estimation E_{join_i} considers the current result as samples of the complete result. E_{join_i} is computed by dividing the current result size by $x_i * y_i$, where x_i and y_i are the fractions of processed data size from the two joining datasets. QPJ provides different selectivity estimation methods for spatial join and equi-join. We apply the method in [10] to compute equi-join selectivity estimation and apply Geometric Histograms [1] to compute spatial join selectivity estimation. The selectivity estimation $E_{selectivity}$ will only be computed once and E_{join_i} will be recomputed with the new join results in each round. The first round estimated ground truth is $\hat{G}_1 = E_{selectivity}$ and the last round estimated ground truth $\hat{G}_s = E_{join_s}$. The importance factor of $E_{selectivity}$ is larger than the importance factor of E_{join} in the beginning. E_{join}'s importance factor becomes larger with more and more data being processed.

Progressive Output Rate Computation. Result distribution is a widely used metric to evaluate the quality of progressive results [8, 12, 13, 15]. The progressive results can represent the final results well if they have similar result distribution to the complete results [15]. The goal of progressive results builder is to produce progressive results with a similar distribution ratio to the final results. In Theorem 1 [16], we proved that when all partitions have the same result rate, the progressive answer has the best quality. The progressive results builder aims to let each partition have roughly the same estimated output rate ρ_i. ρ_i is the ratio of the current result size nO_i (result update to batch i) to the estimated ground truth $n\hat{GT}$, where $\rho_i = nO_i/n\hat{GT}$. The optimal ρ is the minimum ratio among all partitions, ensuring that each partition has the same output result ratio. However, a partition with a small ratio might block other partitions. To guarantee users receive enough amount of results, the progressive results builder allows every partition to not strictly follow the exact same ρ_i. It uses a greedy algorithm [16] to compute the boost output rate ρ^*:

$$\rho^* = \frac{\rho_{i,j} + \ldots + \rho_{i,j}}{j - k\varepsilon}, \quad (2)$$

where k is number of partition, $\varepsilon \geq 0$ is the error bound, and $\rho_{i,j}$ represents the true output rate of partition j up to batch i.

Progressive Output Selection. With the boost output progressive rate ρ^* and estimated ground truth, QPJ computes the size of output results and outputs the progressive results based on computed size. QPJ adopts a two-level sampling method to achieve a finer control for picking results to output. It first partitions data into big partitions which are called coarser-level partitions. Then it further splits each big partition into small partitions which are called finer-level partitions. The progressive rate and output result size are computed based on the coarser-level partition following Equation 1 and Equation 2. In each coarser-level partition, we further compute the result ratio of each finer-level partition.

QPJ performs weighted without replacement sampling strategy. If the output results size is close to the join results size, we apply the sampling method to pick the join results that are temporary hold. If the output results size is small, we apply the sampling method to pick the output results. A more detailed design is introduced in our previous work [16] (Section 5.2).

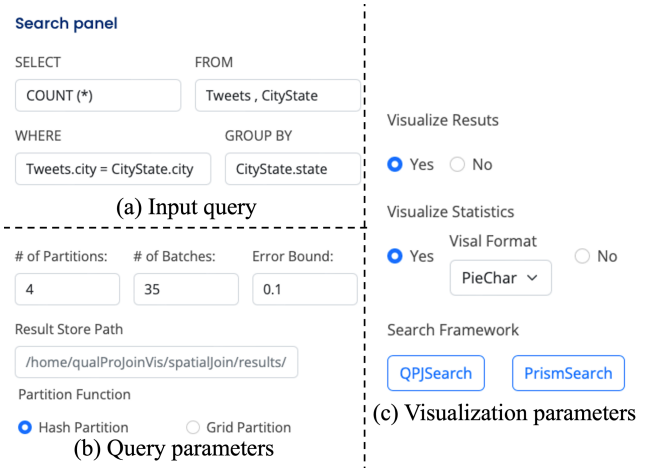

Figure 3: Running a progressive equi-join query.

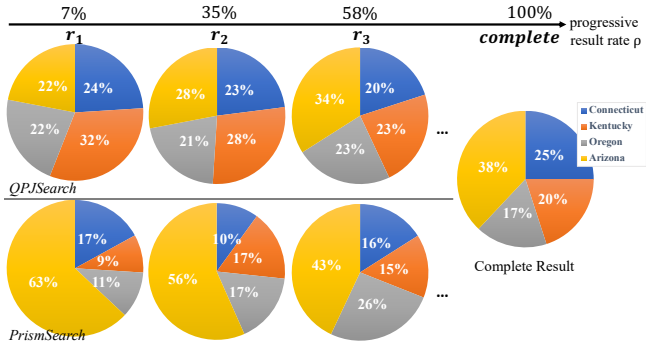

Figure 4: Comparing data distribution of progressive results and complete results: QPJ versus baseline method Prism.

3 DEMONSTRATION SCENARIOS

In this section, we use a progressive equi-join example to demonstrate how to use QPJVis and compare the progressive results computed by different frameworks. We also provide other spatial and non-spatial datasets to enable users to perform progressive spatial join and equi-join queries.

3.1 Running Equi-join Example

Assume a user wants to know how many tweets are posted in each state. The user can apply the following join query with GROUP BY aggregation on the state:" SELECT COUNT(*) FROM Tweets, CityState WHERE Tweets.city = CityState.city GROUP BY CityState.state". In Figure 3, we show the screenshots of the search panel. Users can refer to Figure 3 (a) to enter the input query. In this example, we assume the number of partitions equals 4 and choose the hash partition as the partition function. We set the number of progressive batches to 10. QPJVis will process this query into 10 batches and return 10 progressive results. Additionally, we can assign the error bound for boosting the output rate and assign the result file path. Users can refer to Figure 3 (b) to assign these parameters. If we click QPJSearch, QPJVis will apply QPJ framework to process the given query. Clicking PrismSearch, QPJVis will apply Prism framework to process the given query. The Prism framework is an input control framework. It partitions the input datasets based on the join key and ensures that the input data of each partition follows the same input processed rate. Prism framework returns all the progressive results to the user. Users can refer to Figure 3 (c) to adjust the visualization parameters and choose the progressive processing framework. If users assign an "Error Bound" ε, QPJVis will compute boost output rate ρ^* to produce more output results. Assume we want to visualize the result statistics by pie charts.

3.2 Different Frameworks Comparison

In the search panel, users can process progressive queries by QPJ or baseline method Prism. To compare the quality of the progressive results, we compute the partitioned result distributions of the complete result and progressive results of the two frameworks in Figure 4. We further visualize the result statistics in pie charts. The complete result contains 546 tweets from Arizona state, 360 tweets from Connecticut state, 288 tweets from Kentucky state, and 246 tweets from Oregon state. The total number of results is 1440. In this example, there are four states and the user assigns four partitions so that each partition contains one state. The partitioned result distribution is the result ratio of Arizona = 546/1440 = 0.38 (in yellow), the result ratio of Connecticut = 360/1440 = 0.25 (in blue), the result ratio of Kentucky = 288/1440 = 0.20 (in orange), and the result ratio of Oregon = 246/1440 = 0.17 (in grey).

In the first round, Prism solution gets 63 tweets from Arizona, 17 tweets from Connecticut, 9 tweets from Kentucky, and 11 tweets from Oregon. It returns all the results to the user. Let's use r_{state} to represent the result ratio of a state. The output result distribution is $r_{Arizona} = 63/100 = 0.63$, $r_{Connecticut} = 17/100 = 0.17$. $r_{Kentucky} = 9/100 = 0.09$, and $r_{Arizona} = 11/100 = 0.11$. QPJ solution returns 6 tweets from Arizona, 7 tweets from Connecticut, 9 tweets from Kentucky, and 6 tweets from Oregon. The output result distribution is $r_{Arizona} = 6/28 = 0.22$, $r_{Connecticut} = 7/28 = 0.24$. $r_{Kentucky} = 9/28 = 0.32$, and $r_{Arizona} = 6/28 = 0.22$.

We can compute the mean absolute percentage error (MAPE) of the two progressive results and use the MAPE error to reflect the quality of the progressive results. The MAPE error computes as follows: MAPE $= \sum_{i=1}^{k} \left| \frac{r_G - r_o}{r_G} \right| / k$, where r_G is ground truth result distribution, r_o is the progressive result distribution, and k is the number of partitions. The MAPE of the Prism is $(\frac{0.63-0.38}{0.38} + \frac{0.17-0.25}{0.25} + \frac{0.09-0.20}{0.20} + \frac{0.11-0.17}{0.17})/4 = 0.12$. The MAPE of the QPJ is $(\frac{0.22-0.38}{0.38} + \frac{0.24-0.25}{0.25} + \frac{0.32-0.20}{0.20} + \frac{0.22-0.17}{0.17})/4 = 0.09$. The error of QPJ results is smaller than the error of Prism result so that the quality of the QPJ results is better than the quality of Prism result. By clicking the "Next Batch" button in the result panel (shown in Figure 1), the user can access the progressive results of the following rounds. The partitioned result distribution of the progressive results will become closer to the complete results as the system processes more and more data.

In progressive query processing, the main caveat is that progressive results may not accurately reflect the complete result. Poor quality progressive results can negatively impact further analyses and mislead data scientists to have cognitive biases [12]. In this example. if the user does not have prior knowledge, she or he will draw a wrong conclusion by Prism's progressive results. The user might consider that Arizona state has more results than the sum of other states. On the other side, if the user has prior knowledge, she or he will spend more time waiting for accurate results. In contrast, QPJ does not mislead users like Prism. Progressive results produced by QPJ more closely resemble the complete result than pie charts produced by Prism.

In addition to relational datasets and equi-join query processing, QPJVis also includes spatial datasets to enable users to execute spatial join queries. The following example is a spatial join query:" SELECT obj.range FROM Park, Water WHERE Park.range overlap Water.range". If QPJVis detects the keyword "overlap" in the WHERE clause, the system will perform spatial data processing.

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] Ning An et al. 2001. Selectivity estimation for spatial joins. In *ICDE*. 368–375.
[2] Lars Arge et al. 1998. Scalable sweeping-based spatial join. In *VLDB*, Vol. 98. Citeseer, 570–581.
[3] Liming Dong et al. 2020. Marviq: Quality-Aware Geospatial Visualization of Range-Selection Queries Using Materialization. In *SIGMOD*. 67–82.
[4] Chandramouli Badrish et al. 2013. Scalable progressive analytics on big data in the cloud. *PVLDB* 6, 14 (2013), 1726–1737.
[5] Ding Mengsu et al. 2021. Progressive Join Algorithms Considering User Preference.. In *CIDR*.
[6] Moritz Dominik et al. 2017. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*. 2904–2915.
[7] Procopio Marianne et al. 2019. Selective wander join: Fast progressive visualizations for data joins. In *Informatics*, Vol. 6. MDPI, 14.
[8] Wee Hyong Tok et al. 2008. A stratified approach to progressive approximate joins. In *EDBT*. 582–593.
[9] Zhao Zhuoyue et al. 2020. Efficient join synopsis maintenance for data warehouse. In *SIGMOD*. 2027–2042.
[10] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.
[11] Oje Kwon and Ki-Joune Li. 2011. Progressive spatial join for polygon data stream. In *SIGSPATIAL*. 389–392.
[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] The OpenLayers Dev Team. 2006. OpenLayers. https://openlayers.org
[15] Wee Hyong Tok and Stéphane Bressan. 2013. Progressive and approximate join algorithms on data streams. In *Advanced Query Processing: Volume 1: Issues and Trends*. Springer, 157–185.
[16] Xin Zhang and Ahmed Eldawy. 2023. Less is More: How Fewer Results Improve Progressive Join Query Processing. In *SSDBM*. 1–12.