Less is More: How Fewer Results Improve Progressive Join Query Processing

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

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

ABSTRACT

With the requirements to enable data analytics and exploration interactively and efficiently, progressive data processing, especially progressive join, became essential to data science. Join queries are particularly challenging due to the correlation between input datasets which causes the results to be biased towards some join keys. Existing methods carefully control which parts of the input to process in order to improve the quality of progressive results. If the quality is not satisfactory, they will process more data to improve the result. In this paper, we propose an alternative approach that initially seems counter-intuitive but surprisingly works very well. After query processing, we intentionally report fewer results to the user with the goal of improving the quality. The key idea is that if the output is deviated from the correct distribution, we temporarily hide some results to correct the bias. As we process more data, the hidden results are inserted back until the full dataset is processed. The main challenge is that we do not know the correct output distribution while the progressive query is running. In this work, we formally define the progressive join problem with quality and progressive result rate constraints. We propose an input&output quality-aware progressive join framework (*QPJ*) that (1) provides input control that decides which parts of the input to process; (2) estimates the final result distribution progressively; (3) automatically controls the quality of the progressive output rate; and (4) combines input&output control to enable quality control of the progressive results. We compare *QPJ* with existing methods and show *QPJ* can provide the progressive output that can represent the final answer better than existing methods.

CCS CONCEPTS

• **Information systems → Query representation**.

KEYWORDS

Progressive processing, progressive result quality control, progressive equi-join, progressive spatial join

ACM Reference Format:
Xin Zhang and Ahmed Eldawy. 2023. Less is More: How Fewer Results Improve Progressive Join Query Processing. In *35th International Conference on Scientific and Statistical Database Management (SSDBM 2023), July 10–12, 2023, Los Angeles, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3603719.3603728

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

SSDBM 2023, July 10–12, 2023, Los Angeles, CA, USA

ACM ISBN 979-8-4007-0746-9/23/07.
https://doi.org/10.1145/3603719.3603728

1 INTRODUCTION

Data analysis on large datasets takes minutes or even hours to complete due to big data volume and computation complexity [34]. The challenge becomes even greater when analysis involves multiple big datasets and expensive queries, such as join queries. To address this issue, progressive query processing has emerged as a popular tool [3, 4, 13–15, 35]. Progressive processing splits large datasets into small batches and processes each data batch progressively. Each progressive computation cycle takes a few seconds to keep the users engaged and active [38]. Users can keep examining intermediate results without having to wait for the entire computation to complete on the whole dataset [34]. Progressive answers also enable users to start further processing early on. Data visualization [7, 14, 35, 44] and aggregate queries [11, 15] are frequently employed methods for analyzing progressive answers. Fig.1 shows an example of progressive equi-join over two datasets and visualizes bar charts from the progressive answers in the result set. In addition to data visualization and aggregate queries, synopses maintenance [15, 24] is another form of further processing over progressive results.

Existing progressive processing systems rely on main two techniques to ensure the quality of progressive results: (1) Optimizing the progressive input before the query processing: these systems [11, 12, 14, 15, 18, 38] control the input that goes into query processing. They manipulate the progressive input based on predefined input computation goals. The goals can be the number of items in progressive input, data distribution of progressive input, and preference score function. (2) Optimizing results during query processing: a process ingests and processes more input until the output reaches a desired quality bound [7, 24, 35]. They manipulate the query processing based on result quality goals, for example, error bound or sample strategies. Both solutions can be viewed as *input control* strategies.

The first solution directly returns results and ignores further optimization, which can lead to misleading results. We use the example in Fig.1 to discuss this limitation. In Fig.1, we join Tweets dataset with CityState dataset by existing input control solution *Prism* and our proposed solution *QPJ*. And we further aggregate progressive results in pie charts to compare the progressive results produced by the two solutions. Progressive solutions progressively process the query and data in multiple rounds. *QPJ* and *Prism* [11] produce different pie charts at $r_1, r_2,$ and r_3. The main caveat is that progressive results may not accurately reflect the complete result. Poor quality progressive results can negatively impact further

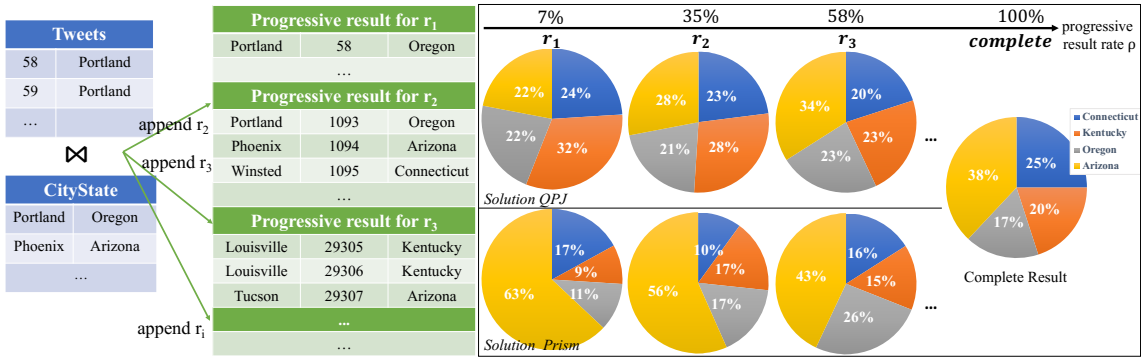


Figure 1: Right side: Progressively join the Tweets with the CityState datasets. The progressive results are continually appended to the result set. Left side: Visualize the first three round results. The upper results compute from input&output quality control framework *QPJ*. The downside results compute from purely input quality control framework *Prism*.

analyses and mislead data scientists to have cognitive biases [34]. For instance, at round r_1, users without prior knowledge will draw a wrong conclusion by *Prism*'s progressive results that Arizona state has more results than sum of other states. On the other side, users with prior knowledge will pay more time waiting for accurate results. In contrast, *QPJ* does not mislead users like *Prism*. Pie charts produced by *QPJ* more closely resemble the complete result than pie charts produced by *Prism*. In the example above, we use similarity to the complete results as a metric to evaluate the quality of progressive results. It is a widely used metric in the literature [13, 14, 18, 34, 35, 38, 39]. We summarize other quality metrics in existing systems in Section.2.

The second solution takes longer 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 [7, 24] instead of exact answers. Therefore, a better strategy to control the quality of progressive results is needed, which provides verified progressive results without sacrificing the advantages of progressive processing.

Another limitation in existing progressive systems are their limited applicability to different data processing tasks. In Section 2, we review several querying processing systems. Many systems are designed for either specific data types or specific query types, as different data types and query types require different algorithms to process. The data analytic tasks across a variety of real-world applications. Therefore, there is a need for a general and lightweight system that can handle both spatial and relational data.

To address the limitations in existing systems, we proposed *QPJ*, a quality-aware framework for equi-join and spatial join queries. *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 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 inserts 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 one by one when the size of the temporary hold result is large. And it uses reverse weighted sampling to filter out results when the size of the temporary hold result is small. Additionally, *QPJ* adopts a dynamic strategy to estimate complete result distribution.

In summary, we make the following contributions:

- We introduce an input&output control quality-aware progressive join framework called *QPJ* in Section 3.
- We propose a solution to guarantee both progressive results' quality and progressive results' rate in Section 4 and 5.
- We provide a dynamic result estimation method. It combines two selectivity estimation strategies and dynamically adjusts weights for two estimation strategies in Section 5.
- We design weighted sampling and reverse weighted sampling to construct progressive output results in Section 5.
- We compared *QPJ* with existing progressive join solutions and showed *QPJ* can return progressive results better than the existing solutions in Section 6.

2 RELATED WORK

2.1 Data Analytics and Data Processing System

Table1 summarizes several recent systems which are designed for data analytics and progress processing systems. We classify existing systems according to four essential aspects of data analytics which are represented by the columns. Each row in the table represents one of the existing systems and our proposed system is highlighted at the end. There are still lots of good data analytics systems, e.g., GeoSparkViz [44] Cloudberry [30], Voyager [19], Visclean [31], NeuralCubes [20], LIBKDV [5], Tabula [22], etc, and join processing frameworks, e.g., PSJ [39], BiStream [16], Poet [33], etc. The main focuses of those systems cannot cover most of the columns in the table, therefore, we did not list them in Table1.

Results return. To let users examine the results quickly, SJoin [24] computes a small sample of the large data, and Marviq [7] returns approximate results. Progressive processing systems enable users to check the intermediate results of complex computations early without waiting for the computation to finish [34]. In Table 1, most of the systems SelectiveWanderJoin [15], IncVisage [35], Pangloss [14], Prism [11], ContourJoin [12], and RRPJ [18, 38] are return progressive results. Those systems compute the results progressively and continuously return the partial results until the computation is finished or users stop the query. *QPJ* is designed for large data processing to enable user interactions in the computation.

System	Results Return	Quality	Processing Control	Join Support
SelectiveWanderJoin [15]	Progressive	None	Input - Not quality control	equi-join
IncVisage [35]	Progressive	Quality aware	Input - During processing	×
Pangloss [14]	Progressive	Quality aware	Input - Before processing	×
Prism [11]	Progressive	Quality aware	Input - Before processing	equi-join
ContourJoin [12]	Progressive	Quality aware	Input - Before processing	equi-join
SJoin [24]	Approximate	Quality aware	Input - During processing	equi-join
Marviq [7]	Approximate	Quality aware	Input - During processing	×
RRPJ [18, 38]	Progressive	Quality aware	Input - Before processing	equi-join and spatial join
QPJ	Progressive	Quality aware	Input&Output - Before and after processing	equi-join and spatial join

Table 1: Summary of data analytics and data exploration systems.

Result quality and quality control. We summarize the three quality control strategies into two categories: input control and output control, by whether the frameworks optimize results after query processing. SJoin [24] maintains the sample of join results for dynamically updated data and does not provide the quality metric for the sample. They continually refine the join sample when the input data dynamic changes. SelectiveWanderJoin [15] lets users examine the intermediate results or let users evaluate the results. SelectiveWanderJoin adjusts the progressive input based on users' feedback and does not provide the default quality metric. We categorize SelectiveWanderJoin into the input control system but not the input quality control system.

IncVisage [35] computes the distance between the progressive results and the estimated results. They incremental split and produce the visualized results. The visualized results are segments. They continuously split segments if the distance to the estimated results is below the quality bound. Marviq [7] computes the quality by the similarity between the approximate results to the precise results. They propose the Jaccard Similarity for scatterplot visualization and a more general mean squared error quality function for scatterplot visualization and heatmap visualization. Both IncVisage and Marviq control the result quality during the query processing, their algorithms are designed specifically for figure results.

Pangloss [14] follows the Sample+Seek [6] method to compute the sample from the whole dataset as the progressive inputs. The Sample+Seek method optimizes the expected distance between the normalized distributions of the approximate answer and the precise one [14]. Pangloss also computes the confidence interval through the worst-case estimates. Prism [11] batch the input data based on pre-computed partition and batch size. In each round, the progressive input of each partition strictly follows the same batch id. Prism returns progressive results in the same form and semantics as the final results and also adjusts the granularity of the progressive input regarding user feedback. ContourJoin [12] requires a pre-processing step to rank the input data by user preference score in descending order. The ContourJoin algorithm to divides the whole data into small batches by contour lines. The contour line represents the preference score bound of each data batch. ContourJoin evaluates and ranks the quality of progressive results by the preference score of the result tuple and returns results progressively in descending order. RRPJ [18, 38] evaluates the quality of the results in terms of result distribution. They use the result distribution to evaluate the quality of the system. Pangloss, Prism, ContourJoin, and RRPJ are input quality control systems by manipulating progressive inputs.

QPJ applies *input&output control* strategy for the results' quality. It batches and partitions the input data by pre-defined goals. Besides, *QPJ* also adjusts progressive output size to control the distribution of progressive outputs. This makes *QPJ* more suitable for data exploration, where users are generally not aware of the data semantics and are not able to tune the system manually.

Processed queries. IncVisage [35] and Marviq [7] are designed for processing queries on a single table. However, data analytics tasks are complex and involve multiple datasets. IncVisage and Pangloss [14, 20] can handle aggregate queries on multiple tables. SelectiveWanderJoin [15], Prism [11], ContourJoin [12], and SJoin [24] can process equi-join queries. RRPJ [18, 38] have two separate frameworks to deal with both equi-join and spatial join queries.

To enable flexible future analytics of the query results, *QPJ* provides both aggregate results, e.g. results count, and the actual query results to the user. Moreover, *QPJ* is designed for handling complex queries, equi-join, and spatial join.

2.2 Sampling Methods

Tao [37] summarizes the algorithmic techniques for independent sampling. Tao lists several sampling techniques, such as weighted sampling and weighted range sampling, and also introduces methods to construct samples. SJoin [24] constructs uniform random samples over join results. BlinkDB [17] constructs stratified samples to compute approximate results for online aggregation queries. *QPJ* adopts a weighted without replacement sampling strategy.

3 *QPJ* FRAMEWORK OVERVIEW

In this section, we describe *QPJ* framework and introduce how it progressive processes join queries.

3.1 *QPJ* architecture

QPJ is a quality-aware progressive query processing framework. Fig.2 shows its architecture. It takes input parameters from users and returns progressive results back to users. *QPJ* produces the quality preserved progressive results in many rounds. The number of rounds is given by the user, we call the data of each round as *batch*. The *QPJ* consists of three components: partitioners, processors, and a progressive results builder. They are drawn into different colors.

In the following, we introduce the usage of *QPJ* by the join queries because it involves multiple datasets. Single dataset processing follows a similar process as join queries processing. Assume there are two input data sets S and R. The user gives a join query,

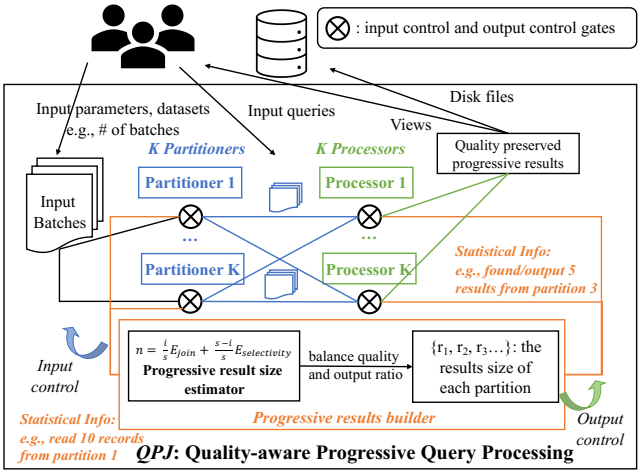


Figure 2: *QPJ* architecture.

the number of partitions k, and the number of progressive computation rounds s. *QPJ* divides datasets S and R into disjoint k partitions and collects the statistical information used for batching and result size estimation. The detailed introduction of batching and partition will be discussed later. 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* starts with loading the first batch and sending it to the k partitioners. Each partitioner reads the data, stores the data belonging to its own partition for its own processor, and sends the rest of data to other processors. Then, k processors answer the input query. Circle black symbols represent the input control and output control gates. Progressive inputs pass input control gates, send to processors, and return the statistical information of each input batch to the progressive results builder.

In each round, the partitioners and processors let the progressive results builder (in orange) know the statistical information of the input data and query results. The progressive results builder estimates the result size and computes the output result size for each partition. Based on the given output size, the k processors output part of results and store the rest results in memory. Progressive results and their statistical information pass the output control gates and send to the progressive results builder. After the progressive results builder returns the number of output results back to the output control gates, the gates let processors release the progressive results based on the output results size.

3.2 Progressive query processing *QPJ*

The progressive results builder and query processing process enable *QPJ* to provide progressive results with good quality. In this subsection, we begin by discussing the high-level design. Subsequently, we introduce how *QPJ* batches and partitions the datasets.

3.2.1 Progressive results builder. The progressive results builder enables *QPJ* automatically control the input and output of each round's progressive computation. The goal of the progressive results builder is to return enough amount of progressive results with good quality. To finish this task, two problems need to be solved: ground truth estimation and finding the optimal progressive output size. Note that the exact ground truth is unknown during the progressive computation. We can only compute the estimated ground truth $\hat{G}$. During the progressive processing, the progressive results builder collects statistics of output results from each partition to compute the estimated ground truth and returns the best partial answer. To better estimate the ground truth, progressive results builder adopts multiple estimation methods and combines them dynamically by different importance. The progressive results builder makes sure each partition has the roughly same estimated output rate ρ_i. The output rate ρ_i is the ratio of the current result size to the estimated ground truth. The progressive results builder computes the best result rate of each partition and computes the output result size of each partition by timing estimated ground truth with result rate. The progressive results builder tries to produce progressive results with a similar distribution ratio to the final results. Therefore, the progressive results can represent the final results well, and we think these types of results with "good quality" [38].

To guarantee users receive enough amount of results, the progressive results builder allows every partition not strictly follows the exact same ρ_i. We provide a greedy solution to increase the output rate, the algorithm will be discussed in Section5. Besides, the input control also helps increase the output rate. As we introduced before, *QPJ* computes the number of processed data in each round in the beginning. When the output rate is not increased significantly, the progressive results builder will increase the number of processed data in the following rounds. Therefore, *QPJ* is an auto-control framework without manual adjustments. To the best of our knowledge, *QPJ* is the first quality-aware auto input and output control progressive processing framework.

3.2.2 Query processing. *QPJ* is a flexible framework and can plug in any query processing algorithm. To process the equi-join query, *QPJ* applies hash join algorithm [25]. To hand the spatial join query, *QPJ* applies the Plane Sweep algorithm [2]. 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 current batches. It progressively produces the whole query results. Section5 describes the progressive processing with more details.

3.2.3 Batching and data partition. **Batching.** Following the above example, *QPJ* processes S and R in s rounds. Based on the input integer s and the size of data sets, *QPJ* can efficiently compute the size of s input batches by any split function. In round i, *QPJ* will provide the progressive answer up to batch i. The simplest way to create batches is to split each partition into equal-sized batches. However, as more and more batches are buffered, the later rounds will process more data than the early round if apply equal-size split method. The unbalanced workload triggers irregular processing intervals. If the processed data workload in each processing unit is balanced, users can check the progressive results in regular fixed intervals. The workload balance problem is also an interesting research direction that can be worked on [10, 21]. However, the main contribution of this work is not workload balance. Besides the equal split solution, we also provide a balanced solution. We use a balance function to make sure 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.

Data partition. *QPJ* provides hashing partition and grid partition. The hashing partition [11, 16, 18] 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 [33, 36, 38]. It divides the input data space by equal-size grid cells and hashes each grid cell to a different partition. For relational data, each row belongs to only one partition (single-assign). While spatial vector objects are points, lines, and polygons. A polygon might belong to multiple partitions if this polygon overlaps with multiple partitions [32] (multi-assign). Because partitions are disjoint with each other, the processor i only needs to join the partition i of dataset1 with the partition j of dataset2. For equi-join, following the full partition-wise joins[1], if the two records do not have the same joined attribute, they won't be a query answer. For spatial join, the fully contained objects in one partition do not overlap with the fully contained objects in another partition. The partitioning stage helps *QPJ* save the computation overhead on join operations.

4 THE QUALITY OF PROGRESSIVE RESULTS

In this section, we introduce the mathematical definition of progressive results' quality. Unlike existing methods [11, 14, 18, 35, 38] only focus on result distribution, *QPJ* also considers the output rate of the progressive results. This makes sure users get enough results to help with further analytics on the query results.

4.1 Symbols and notations

Symbols	Description
S, R	Input dataset S and R.
Q, Ans_Q	Join query Q and its answer Ans_Q
ans_i n_i	ans_i is progressive results up to input batch i n_i is the number of result tuples in ans_i, $n_i = \|ans_i\|$
nGT_j	The **ground truth** result size of partition j
$ans_{i,j}$ $n_{i,j}$	$ans_{i,j}$ is progressive results of partition j update to batch i, $n_{i,j} = \|ans_{i,j}\|$
$ansO_{i,j}$ $nO_{i,j}$	$ansO_{i,j}$ is **output** progressive results of partition j for round i, $nO_{i,j} = \|ansO_{i,j}\|$
ρ_i	The result rate of round i, $\rho_i = \|ans_i\|/\|Ans_Q\|$.
$r_{i,j}$	The ratio of partition j over all the partition up to batch i. $r_{i,j} = n_{i,j}/(\sum_{j=1}^{k} n_{i,j})$
$e_{i,j}$	The output error of partition j in round i.
E_i	The average error of the progressive answer ans_i.

Table 2: Summary of the symbols used in this paper.

Table2 lists the symbols used throughout this paper. Assume the inputs are two datasets S and R, a join query Q, an integer s ($s \geq 1$) which is the number of rounds, and the error bound $\varepsilon(\varepsilon \geq 0)$. Both S and R contain many records, like rows or spatial polygons. For example, row records with row id and string attributes, and spatial polygons with latitude and longitude. The join query Q can be an equi-join query or a spatial join query. The outputs from the problem are s output progressive answers $\{ansO_1, ansO_2, ..., ansO_s\}$ which satisfy the error bound ε. The ε is the average error bound of progressive results in each round. Ans_Q is the query answer to the input join query Q, where $Ans_Q = \bigcup_{i=1}^{s} ans_i$.

The framework divides the input datasets into s batches, one batch per round. In each round, the framework processed one batch of data from S and one batch from R. In round i, the progressive answer ans_i is computed from data up to batch i. To evaluate the similarity between the progressive results to the final answer, we partition input datasets into k disjoint partitions and compute the result distribution from each partition. When the progressive results are actual result tuples instead of aggregation, most of the existing works [14, 18, 38] evaluate result quality through distribution.

Definition 1. ***Progressive results*** *ans_i are the results that are computed from input batches $s_1, ..., s_i$.*

Definition 2. *Based on partition function, the ans_i can be represented by **partitioned progressive results**: $ans_i = \{ans_{i,1}, ..., ans_{i,k}\}$, where $ans_{i,j}$ is the results in partition j up to batch i.*

Let n represent the number of results. In round i, we found $n_i = \langle n_{i,1}, ..., n_{i,k} \rangle$ results. Based on definition 2, $n_{i,j} = |ans_{i,j}|$ denotes the number of results found in partition j up to batch i.

Definition 3. ***Output progressive results*** *$ansO_i = \{ansO_{i,1}, ..., ansO_{i,k}\}$ are results outputted in round i. For each partition j, pick a random sample $ansO_{i,j}$ from $ans_{i,j}$. The picked $ansO_i$ satisfies the error bound ε.*

Let nO_i denote the size of the output progressive answer size. $nO_i = \langle nO_{i,1}, ..., nO_{i,k} \rangle$, where $nO_{i,j} = |ansO_{i,j}|$ and $0 \leq nO_{i,j} \leq n_{i,j}$. Let nGT represent the ground truth result size, which is the total number of results if s batches. $nGT = \langle nGT_1, ..., nGT_k \rangle$, where nGT_j denotes the result size in partition j. We use estimated ground truth during the progressive computation. The detailed discussions of estimate ground truth are in Section 5.1.

Definition 4. ***Progressive result rate*** *ρ_i denotes the result rate of output progressive answer ans_i up to batch i. $\rho_{i,j}$ denotes the result rate of partition j up to batch i. It is computed by diving the ground truth result size nGT_j by the output result size $nO_{i,j}$.*

$$\rho_{i,j} = \frac{nO_{i,j}}{nGT_j}, \rho_i = \frac{\sum_{j=1}^{k} nOi,j}{\sum_{j=1}^{k} nGT_j} \tag{1}$$

Next, let's define the result distribution and result quality.

Definition 5. *The **accurate result distribution** is computed based on the ground truth result size. $rG = \langle rG_1, ..., rG_k \rangle$, where $rG_j = nGT_j / \sum_{j=1}^{k} nGT_j$*

Definition 6. *The **output result ratio** $rO_i = \langle rO_{i,1}, ..., rO_{i,k} \rangle$, where $rO_{i,j} = nO_{i,j} / \sum_{j=1}^{k} nO_{i,j}$.*

rG_j and $rO_{i,j}$ are result ratio of partition j. Based on the definition, we have $\sum_{j=1}^{k} rG_j = 1$ and $0 \leq rG_j \leq 1$.

Definition 7. *E_i denotes the **average error** of output progressive answer of round i. $e_{i,j}$ denotes the **error** between output result ratio $rO_{i,j}$ and ground truth result ratio rGT_j of partition j.*

$$e_{i,j} = \frac{|rGT_j - rO_{i,j}|}{rGT_j}, E_i = \frac{\sum_{j=1}^{k} e_{i,j}}{k} \tag{2}$$

[1]Full Partition-Wise Joins: https://docs.oracle.com/database/121/VLDBG/GUID-5279BF41-41BF-4F87-A64E-2AA58C22BD61.htm

The output progressive result with smaller E_i means its result distribution is close to the ground truth (final) result distribution. Therefore, it can better represent the final results.

Example 1. *In Fig.1, we join Tweets with CityState: SELECT * FROM S,R WHERE S.city = R.city.*

The complete result contains 546 tweets from Arizona (A) state, 360 tweets from Connecticut (C) state, 288 tweets from Kentucky (K) state, and 246 tweets from Oregon (O) state. The total number of results is 1440. The ground truth result distribution is $rGT_{Arizona} = 546/1440 = 0.38$, $rGT_{Connecticut} = 360/1440 = 0.25$, $rGT_{Kentucky} = 288/1440 = 0.20$, *and* $rGT_{Oregon} = 246/1440 = 0.17$.

Definition 8. ***Quality-aware progressive answer computation problem****: given input dataset S and R, join query Q, the number of rounds s ($s \geq 1$), and the error bound ε($\varepsilon \geq 0$), the goal is to return the progressive answers which the maximum result rate ρ_i and also satisfy the error bound. The formal problem statement is:*

$$\max \sum_{j=1}^{k} nO_{i,j}, \text{subject to } \frac{\sum_{j=1}^{k} e_{i,j}}{k} \leq \varepsilon. \tag{3}$$

Please notice that we evaluate result quality by results distribution. More partitions enable finer control over the result quality. However, with more partitions, the result estimation and error computation are more expensive. There is an implicit trade-off between the finer quality control of progressive results and the efficiency to compute the progressive results.

4.2 Existing solutions

Existing solutions RRPJ [18, 38], Prism [11], and ContourJoin [12] maximum the result rate by returning all the found results, which is $nO_{i,j}$=$n_{i,j}$. In this case, they cannot guarantee the progressive answer always satisfied the quality bound ε.

Unlike the existing works, we proposed *QPJ* which returns the progressive answer that satisfies the quality bound ε and maximizes the progressive rate ρ_i as much as possible. To guarantee the error bound, *QPJ* hides part of the results from n_i in memory temporarily and returns the most representative progressive answer.

Theorem 1. *When all the partitions have the same result rate, we have the progressive answer with the best quality.*

When $\rho_{i,1} = \rho_{i,2} = ... = \rho_{i,k}, E_i = 0$.

Proof. By the definition, we have:
(1) $rGT_j = \frac{nGT_j}{\sum_{j=1}^{k} nGT_j}$, (2) $rO_{i,j} = \frac{nO_{i,j}}{\sum_{j=1}^{k} nO_{i,j}}$,
(3) $nO_{i,j} = nGT_j * \rho_{i,j}$.
Then, $rO_{i,j} = \frac{nO_{i,j}}{\sum_{j=1}^{k} nO_{i,j}} = \frac{nGT_j * \rho_{i,j}}{\sum_{j=1}^{k}(nGT_j * \rho_{i,j})}$
$= \frac{\rho_{i,j} * nGT_j}{\rho_{i,j} * \sum_{j=1}^{k} nGT_j} = rGT_j$
$\Rightarrow E_i = \frac{\sum_{j=1}^{k} e_{i,j}}{k} = \frac{\sum_{j=1}^{k}(rGT_j - rO_{i,j})/rGT_j}{k} = 0$
□

Different partition has different result rate ρ. As Theorem1 shows, the more partitions have the same result rate, the better progressive answer we can output to the users.

Example 2. *In the first round of Fig.1, there are 63 tweets from Arizona, 17 tweets from Connecticut, 9 tweets from Kentucky, and 11 tweets from Oregon. Existing solution Prism returns all of them. In Prism's solution,* $\rho_0 = 0.069$.$rO_{0,A} = 0.67$, $rO_{0,C} = 0.17$, $rO_{0,K} = 0.09$, *and* $rO_{0,O} = 0.11$.

The error of Prism solution: $E_0 = (|38 - 63|/38 + |25 - 17|/25 + |20 - 9|/20 + |17 - 11|/17)/4 = 0.52$.

QPJ *returns 6 tweets from Arizona, 7 tweets from Connecticut, 9 tweets from Kentucky, and 6 tweets from Oregon.* $\rho_0 = 0.019$. $rO_{0,A} = 0.22$, $rO_{0,C} = 0.24$, $rO_{0,O} = 0.32$, *and* $rO_{0,O} = 0.22$.

QPJ *error:* $E_0 = (|38 - 22|/38 + |25 - 24|/25 + |20 - 32|/20 + |17 - 22|/17)/4 = 0.42 < 0.52$ *(Prims's solution).*

5 PROGRESSIVE PROCESSING

Note that during the progressive computation, there is no way to get the actual ground truth, we can only compute the estimated ground truth $\hat{G}$. In *QPJ*, the progressive results builder estimates the total number of results $\hat{G}$ based on the current processing progress and current results size. With the estimated ground truth $\hat{G}$, *QPJ* can provide the progressive answer nO_i which satisfies the error bound and maximizes the progressive rate p_i as much as possible.

5.1 Estimated ground truth

Let $n\hat{G}T_i = \langle n\hat{G}T_{i,1}, ..., n\hat{G}T_{i,k} \rangle$, where $n\hat{G}T_{i,j}$ denotes the estimated total number of join results in partition j in round i. The Selectivity Estimation problem is well-studied [25]. There are two popular used ways to estimate the join size:

- Method $E_{selectivity}$ (Equi-join): Consider a join query $R(X, Y) \bowtie S(Y, Z)$ and Y is a key in S and the corresponding foreign key in R, we can estimate the join size $T(R \bowtie S) = T(R)T(S)/\max\{V(R, Y), V(S, Y)\}$, where $T(R)$ is the number of tuples in R, $V(R, Y)$ is the number of distinct Y values in R. [2]
- Method $E_{selectivity}$ (Spatial join): We construct Geometric Histograms [1] to estimate the join size of spatial join.
- Method E_{join}: Existing works [26–28] estimate the result size based on sample result size. We can get $x_i * y_j$ fraction of the final results by joining x_i fraction of the dataset1 with y_j fraction of the dataset2.

In Section6, we test the precision and efficiency of the estimation by applying $E_{selectivity}$ only and applying E_{join} only. The accuracy of $E_{selectivity}$ performed better than E_{join} in the beginning and E_{join} became more accurate than $E_{selectivity}$ with more and more data processed. Because $E_{selectivity}$ requires extra computation overhead and statistical information maintenance, only applying E_{join} ran faster than applying $E_{selectivity}$ only. To make use of the advantages of the two estimation methods, progressive results builder combines the $E_{selectivity}$ with E_{join} in each round by linear weight function. The weights of two estimations are changed.

The progressive results builder collects the necessary statistical information and computes the value of selectivity estimation $E_{selectivity}$ only once in the partition phase. Any selectivity estimation method can be plugged into *QPJ* to compute the $E_{selectivity}$.

[2] The book [25] mentioned two assumptions: Containment of Value Sets and Preservation of Value Sets. If the query satisfies the above two assumptions, then we can estimate the join size based on the equation. If Y is a key in S and the corresponding foreign key in R, the query satisfies the two assumptions.

Based on the batch function introduced in Section3-Batching and data partition, we know the ratio of the current batch size to the total result size. $E_{join_i} = (m1 * m2)/(m1_i * m2_i * n_i)$, where $m1, m2, m1_i, m2_i$ are input data size and batch size, n_i is the result size of round i. Assume the split function divides the data into s batches. The progressive results builder dynamic estimates ground truth result size $E_{dynamic}$ by the following:

$$n\hat{GT}_{i,j} = E_{dynamic} = \frac{i}{s}E_{join_i} + \frac{s-i}{s}E_{selectivity}, \quad (4)$$

where the first estimated ground truth is $\hat{G}_1 = E_{selectivity}$ and the last estimated ground truth $\hat{G}_s = E_{join_s}$.

5.2 Progressive result builder

5.2.1 Optimal progressive rate. With the estimated ground truth $n\hat{GT}_i = \langle G\hat{T}_{i,1}, ..., G\hat{T}_{i,k}\rangle$, we can compute the progressive output rate $\rho_{i,j}$ of each partition j in round i, $\rho_{i,j} = nO_{i,j}/n\hat{GT}_{i,j}$. As Theorem1 claimed, when all the partition has the same progressive rate, we can have a progressive answer with the best quality. To achieve this, we let $\rho_i = \min\{\rho_{i,1}, ..., \rho_{i,k}\}$. Then, the output size $nO_{i,j}$ of partition j is computed by $n\hat{GT}_{i,j} * \rho_i$.

Different progressive rates ρ_i output different numbers of result tuples. If ρ_i is not the minimum process over the k partitions, some partitions cannot return $n\hat{GT}_{i,j} * \rho_i$ result tuples, so $rO_{i,j} \neq r\hat{GT}_{i,j}$. Although taking the minimum ρ for every partition guarantees the progressive answer has the best quality, it also blocks the progressive rate, which let users cannot see enough new results.

Assuming the estimation $n\hat{GT}_{i,j}$ increased sharply in the round i, its new progressive rate is smaller than the previous round's progressive rate. Then $\rho_{i,j}$ would block all of the partitions in round i+1 and *QPJ* cannot output anything. To avoid some partitions that block the output progress rate, *QPJ* enables each partition to have a different progressive rate. It lets most of the partitions return $n\hat{GT}_{i,j} * \rho_i$ number of results, and the rest of partitions can return less than $n\hat{GT}_{i,j} * \rho_i$ number of results.

There is a trade-off between the progressive rate and overall output error. Taking the minimum progressive rate ρ_i over $\{\rho_{i,1}, ..., \rho_{i,k}\}$, every partition can return $n\hat{GT}_{i,j} * \rho_i$ results, then rO_i=$r\hat{GT}_i$ and $\sum_{j=1}^{k} e_{i,j}/k$=0. Boosting the progressive rate ρ_i triggers some partitions that cannot return enough results, which produces a larger overall error. To boost the progressive rate, we can let the overall error reach the maximum error bound ε.

Let's denote the optimal progressive rate up to batch i is ρ_i^*. *QPJ* finds the ρ_i^* based on the greedy Algorithm1. The algorithm first ranks all the progressive rates in ascending order in the list L (lines 1-2). If the ρ_i^* is the smallest progressive rate which is the first one in the list L, the error is 0 based on Theorem1. We maintain the error in the previous round as e_{pre}. In the while-loop (line 5-11), the algorithm greedy increases the progressive rate by trying larger progressive rate. The while-loop ends when the error e is larger than the given error bound ε. Then, the algorithm computes the boost optimal progressive rate based on Theorem2 (line 12-13).

Theorem 2. *The boost progressive rate* ρ^* *is* $\frac{\rho_{i,j}+...+\rho_{i,j}}{j-k\varepsilon}$.

Proof. Assume when idx=j+1, the error becomes larger than the error bound ε. The optimal progressive rate computes as following:

Algorithm 1: Boost the progressive rate

Input: The error bound ε, and the output rate of each partition: $\{\rho_{i,1}, ..., \rho_{i,k}\}$
Output: The boosted output rate ρ^*

1 Rank the $\{\rho_{i,1}, ..., \rho_{i,k}\}$ in ascending order
2 List L = output rate list ranked in ascending order
`// Smallest output rate is ranked first`
3 Boost id $idx = 0$, $\rho^* = L[idx]$
4 $e = 0, e_{pre} = 0$ `// The error of this round and previous round`
5 **while** $e < \varepsilon$ **do**
6 $\quad \rho^*$ = L[++idx] `// greedy increase the output ratio`
7 $\quad e_{pre} = e, e = 0$
8 $\quad$ **for** *i from* 0 *to idx* **do**
9 $\quad\quad e$+ = e_{idx}
10 $\quad$ **end**
11 **end**
12 If e == ε, **return** ρ^*
13 **return** $\rho^* = \frac{\rho_{i,1}+...+\rho_{i,idx}}{idx-k\varepsilon}$ // $e_{pre} < \varepsilon < e$

$$e_{i,j} = \frac{r^*_{i,j} - r_{i,j}}{r^*_{i,j}} = 1 - \frac{r_{i,j}}{r^*_{i,j}} = 1 - \frac{r_{i,j}\sum_{j=1}^{k} n^*_{i,j}}{r^*_i \sum_{j=1}^{k} n^*_{i,j}} = 1 - \frac{n_{i,j}}{n^*_{i,j}}$$
$$= 1 - \frac{n_{1,j}}{\rho^*_i n\hat{GT}_{i,j}} = 1 - \frac{1}{\rho^*_i}\frac{n_{i,j}}{n\hat{GT}_{i,j}} = 1 - \frac{1}{\rho^*_i}\rho_{i,j}.$$
$$\Rightarrow k\varepsilon = \sum_{j=1}^{a} e_{i,j} = 1 - \frac{1}{\rho^*_i}\rho_1 + ... + 1 - \frac{1}{\rho^*_i}\rho_{i,j} = j - \frac{\rho_{i,j}+...+\rho_{i,j}}{\rho^*_i}$$
$$\Rightarrow \rho^*_i = \frac{\rho_{1,j}+...+\rho_{i,j}}{j-k\varepsilon}.$$ □

The complexity analysis. The time cost of Algorithm1 comes from two parts: sort phase and while loop. We use the MergeSort for ranking the progressive rates, its time complexity is $O(n \log n)$. In the while loop, the number of computations in for loop is equal to idx, it starts from 1 to k, where k is the number of partitions. In the worst case, idx equals k, and the worst time complexity of the while loop is $O(k^2)$. In the previous section, we pointed out the implicit trade-off between the finer quality control of progressive results (more number of partitions) and the efficiency to compute the progressive results. Therefore, we won't consider too large k in the real-world case. In the experiment section, we test with $k \leq 30$. Compare with the scale of the join algorithms, $O(k^2)$ is not costly.

Optimal size of the progressive answer. With the optimal ρ_i^*, *QPJ* computes the progressive answer $n_{o,i} = \{\hat{G_{i,1}}\rho_i^*, ..., \hat{G_{i,k}}\rho_i^*\}$. The output progressive answer nO_i satisfies the error bound ε and also has the largest boosted progressive rate ρ^*.

Below is the process to compute the optimal progressive rate for Example 1 and Example 2.

Example 3. *In* r_3 *of Fig.1, we have* $\rho_A = 0.70$, $\rho_C = 0.30$, $\rho_K = 0.40$, *and* $\rho_O = 0.65$. *And the error bound* $\varepsilon = 0.2$.

If ρ_i=ρ_K=0.40, *average error is* $\frac{1-\frac{0.3}{0.4}+1-\frac{0.4}{0.4}}{4} = 0.1875 < \varepsilon$.
If ρ_i=ρ_O=0.65, *average error is* $\frac{1-\frac{0.3}{0.65}+1-\frac{0.4}{0.65}+1-\frac{0.65}{0.65}}{4} \approx 0.23 > \varepsilon$.
Algorithm 1 returns optimal $\rho_2^* = \frac{0.3+0.4}{2-4*0.2} \approx 0.58 > \rho_K$.

5.2.2 Select progressive output results. With the optimal output progressive rate ρ^* and estimated ground truth, *QPJ* computes the

size of output results $\{nO_{i,1}, ..., nO_{i,k}\}$ and outputs the progressive results based on computed size. This brings a new problem that how to choose results to output.

QPJ adopts sampling method to solve this problem. A simple solution is adopted uniform sampling, which randomly picks $nO_{i,j}$ results from partition j. Uniform sampling takes each join result with the same importance. We can also achieve a finer control to pick results to output. *QPJ* uses a two-level partitioning strategy. 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 split process is lightweight. In hashing partition, assume the number of coarser-level partitions is c and the number of finer-level partitions is f. Given an item with hash code h, it belongs to finer partition $(h/c \mod f)$ in coarser partition $(h \mod c)$. The progressive rate and output result size are computed based on the coarser-level partition.

QPJ adopts the weighted sampling method to pick output results of equi-join. In each coarser-level partition, we compute the result ratio of each finer-level partition. *QPJ* takes the result ratio as the weight for each finer-level partition. In the partition phase, *QPJ* computes the $E_{selectivity}$ based on statistical information of the whole dataset. The result ratio is computed by $E_{selectivity}$ of each finer-level partition.

Currently, spatial join output results are picked by uniform sampling. Because the current spatial join algorithm in *QPJ* is designed for spatial polygons and each spatial polygon might overlap with multiple finer-level partitions. A polygon might be picked from one overlapping finer-level partition and not be picked by the other overlapping finer-level partitions. *QPJ* uses the uniform sampling method to avoid this situation happens.

5.2.3 reverse weighted sampling. *QPJ* samples progressive output results by weighted without replacement sampling strategy. Existing sampling over joins works [15, 23, 24, 24, 43] are "direct" methods, which insert sample items to the samples. The direct methods generate samples with a small size from join results with a big size. However, applying the direct method to *QPJ* to generate progressive output results is expensive, because the output results size $nO_{i,j}$ is close to the join results size $n_{i,j}$. Therefore, we design **reverse weighted sampling** strategy for *QPJ* to pick the join results that are temporary hold.

For direct weighted sampling methods, items with higher weights have larger chance to be added to samples. Given a set of weights $\{w_{l-1}, ..., w_{l-k}\}$ of partitions $\{1, ..., k\}$ and the size of all the results n, the direct weighted sampling model M is built by the following:

$$M = \{\{1, ..., 1\}, ..., \{k, ..., k\}\}, where|\{i, ..., i\}| = w_{l-i} * n \quad (5)$$

For each sampling round, the system uniform picks a partition id i from M and adds the item of partition i to the samples.

The reverse weighted sampling is the opposite in those items with smaller weights having large chance to be filtered out from samples. Given a set of weights $\{w_{l-1}, ..., w_{l-k}\}$ of partitions $\{1, ..., k\}$ and the size of all the results n, the reverse weighed sampling model RM is built by the following:

$$RM = \{\{1, ..., 1\}, ..., \{k, ..., k\}\}, where|\{i, ..., i\}| = n - w_{l-i} * n \quad (6)$$

The frequency of partition i in RM is reserved by using total size n minus its direct frequency $w_{l-i} * n$. For each sampling round, *QPJ* picks a partition id i from RM and filters out the item of partition i from the found results. The filtered item will be held in memory and merged with join results in the next progressive round.

5.3 Progressive Processing

Algorithm 2: Progressive Join

Input: Dataset S and R, the query Q, the progressive computation rounds s
Output: progressive answer

1 Collect the statistical information from S and R
2 Compute the batch size based on rounds s
3 **for** *each batch* **do**
4 read and partitions the data
5 ships the partitioned data to Processors
6 sends partition size to Progressive Results Builder
7 **end**
8 **for** *each Processor* **do**
9 filter out $N1$ and $N2$ // N_m is the new batch from dataset i
10 $R = R \bigcup$ Join(N1, N2)
11 $R = R \bigcup$ Join(N1, E2)
12 $R = R \bigcup$ Join(N2, E1) // E_m is the old batches from dataset i
13 sends the number of results to Progressive Results Builder
14 **end**
15 Progressive Results Builder estimates result size $\hat{G}_i$ by Equation (2) and output ratio ρ_i and send to Processors
16 Processors output the progressive answer nO_i

As a general framework to handle both equi-join and spatial join queries, *QPJ* integrates different join algorithms. Since algorithms used to process equi-join and spatial join are different, we designed algorithm *Progressive Join* to integrate equi-join and spatial join processing. It transfers the universal input text format to a set of tuples that can be used by the underlying join algorithms. Algorithm 2 shows the pseudocode.

In each round, *QPJ* finishes three tasks (lines 3-7): (1) Reading one input split and partitioning the data. (2) Shipping the partitioned data to the processors. (3) Communicating with the Progressive Results Builder. The partition strategy is similar to the pre-step: applying grid partition strategy for spatial data and applying hashing partition strategy for non-spatial data. After the reading, the processors send the statistical information of the input split back to the Progressive Results Builder.

Each processor joins data from at least one partition. It's depended on the partition size. The processor can process multiple partitions if all of them are small. Line 8-14 shows the three join steps that happened in each processor. N_m is the new batch from dataset m and E_m are the new batches from dataset m. To process the equi-join query, *Progressive Join* applies hash join algorithm [25]. Any existing equi-join algorithm can replace the current equi-join solution. To hand the spatial join query, *Progressive Join* applies the Plane Sweep algorithm [2]. Like the equi-join solution, any spatial join algorithm can replace the current solution. After the three join steps, *Progressive Join* merges the new batches with existing batches and stores them in memory. Then, Progressive Results

Builder starts to estimate the result size and output ratio (line 13), the estimation happens in the Progressive Results Builder. Finally, the Progressive Results Builder sends the number of outputted results from each partition to each processor, and the processors output the progressive answer nO_i.

The complexity analysis. Algorithm 2 does not add extra overhead to the original join algorithms. It just processes the input data in multiple rounds. Therefore, the time complexity of the algorithm is the same as the join algorithms. *QPJ* applies hash join algorithm [25] for equi-join queries and Plane Sweep algorithm [2] for spatial join queries. The time complexity of the hash join algorithm is $O(M+N)$, and the time complexity of the Plane Sweep algorithm is $O((M+N)\log(M+N))$, where M and N are the sizes of the two input datasets.

6 EXPERIMENT

6.1 Experimental Setup

We conduct the experiments on Intel(R) Xeon(R) CPU E5-2609 v4 @ 1.70GHz and runs CentOS Linux release 7.5.1804 (Core). It has 128 GB RAM and 2×8-core processors.

Datasets. We test *QPJ* by two real-world datasets. We extract the 50 M Geo Twitter data [40] to test the equi-join. Each Twitter contains a geo tag that includes the city information. We extract the location formation from Twitter data for performing join operations between Twitter and location. And use the two spatial datasets OSM2015/lakes [8] and OSM2015/parks [9] to test the spatial join. The OSM2015/lakes dataset contains 7.5 M polygons, and the OSM2015/parks dataset contains around 10 M polygons. We shuffle the datasets based on chi-square distribution [41] and discrete uniform distribution [42].

Test Queries. We join the Twitter data with the city and state. The synthetic of the equi-join query is SELECT Twitter.id FROM Twitter, CityStateInfo WHERE Twitter.city = CityStateInfo.city. There are multiple types of spatial join [29], e.g., intersecting, range query, similar join, etc. We evaluate the spatial join by intersecting spatial join. Given two sets of polygons, find all pairs of intersecting polygons between the two sets [29]. All the join queries are full-history join.

Compared methods. We compare our framework *QPJ* with two progressive input control frameworks in Table 1: Prism [11] and ContourJoin (ConJ) [12]. We pick *Prism* because it is a classical input control framework. Prism process batches from every partition in a synchronized non-decreasing order. Each round guarantees to process the batches with the same batch id and show whatever they found to the users. We pick ConJ because it is a recently proposed progressive processing framework and its quality control strategy is different with *Prism*. To perform ConJ, we generate user preference score for the test data. We consider twitter id as the preference score of Twitter data and consider polygon id as the spatial data. For progressive frameworks *Prism* and ConJ, we evaluate the quality of the progressive results by computing the difference between the final result distribution and the progressive result distribution.

Besides progressive frameworks, we also consider using Spark to perform equi-join and spatial join as the non-progressive baseline. We write the Spark join results on disk and split the result files into a set of partial results. The size of the partial result is the same as the size progressive result of *QPJ*. We compute and compare the quality of partial results of Spark with progressive results of *QPJ*.

6.2 Evaluation Metrics

Quality. We evaluate the quality of progressive results by result distribution similarity to the ground truth. The ground truth is the total number of results in each partition. The quality evaluation metrics come from [38]. We take Kullback Leibler (KL) divergence (KL) and mean absolute percentage error (MAPE) to compute the quality of the progressive partial answers. The Kullback Leibler (KL) divergence measures the differences between the actual and partial results produced. The mean absolute percentage error (MAPE) measures the differences between the ground truth and the partial answers. They are computed as followings:

$$\mathrm{KL} = \sum_{i=1}^{k} r_{G,i} \log\left(\frac{r_{G,i}}{r_{o,i}}\right), \mathrm{MAPE} = \frac{100\%}{k} \sum_{i=1}^{k} \left|\frac{r_{G,i} - r_{o,i}}{r_{G,i}}\right|, \quad (7)$$

where $r_{G,i}$ is ground truth ratio and $r_{o,i}$ is the ratio of the partial answer $n_{o,i}$, and k is the number of partitions.

Efficiency. The efficiency is evaluated by the progressive output rate changed over time.

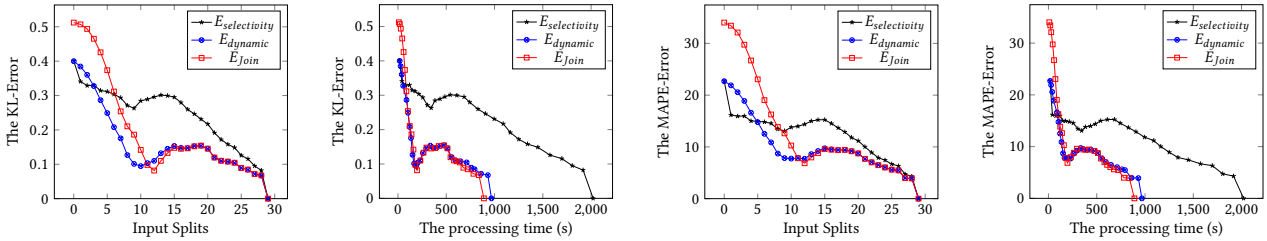
Existing distributed platforms, e.g., Spark and Flink, do not allow communication operations in the middle of transformation operations. Therefore, we did not provide the distributed results in current experiments. In the future, we hope the existing distributed platforms can support enhanced operations, like enabling communication during transformation operations, to meet the requirements of *QPJ*. We built the first version of *QPJ* on top of Spark Streaming. We divided input files into small batches and broke the join operation into two Spark Streaming operations. We computed the join results in the first operation and outputted the progressive results in the second operation. By this design, we let Spark Streaming support *QPJ*, but it slowed down the efficiency a lot. Therefore, we developed *QPJ* in Java and did not deploy it on existing distributed platforms in the current version.[3]

Experiment Plan. We evaluate *QPJ* from six perspectives: (1) The precision of the estimation methods: we compare the accuracy and efficiency of only applying $E_{selectivity}$, only using E_{join}, and the linear combination described by 4. (2) The quality of progressive results computed by *QPJ* and three baseline methods. The quality is evaluated by the quality metrics KL and MAPE. (3) The error bound: given different error bounds, we compare the quality and progressive output rate of *QPJ* with the best baseline method *Prism*, which produces the smallest error among the three baseline methods. (4) Data distribution: for the datasets with different distributions, we compare the progressive results of *QPJ* with baseline methods. (5) The number of progressive rounds: for the different number of rounds, we compare *QPJ* with *OneBatch* on the response time to see the earlier results. (6) The number of partitions: we compare the quality and progressive output rate with the different numbers of partitions. (7) We compare the quality of progressive output results picked by weighted sampling and uniform sampling.

6.3 Quality Experiment results

6.3.1 Estimation Function. We proposed a dynamic estimation function in Sec.5, Eq. 4 ($E_{dynamic}$) , it linear combines selectivity

[3]The code link: https://github.com/xin-aurora/QualityProgressiveJoin.git



(a) KL Error (b) KL Error With Time (c) MAPE Error (d) MAPE Error With Time

Figure 3: The comparison of the accuracy and efficiency of different estimation methods.

estimation method ($E_{selectivity}$) and sample method (E_{join}). Fig.3 compares the three result estimation methods: $E_{selectivity}$ in black, $E_{dynamic}$ in blue, and E_{join} in red. The accuracy of our proposed $E_{dynamic}$ and $E_{selectivity}$ are better than E_{join} in the beginning. As more data be processed, $E_{selectivity}$'s precision decreases, E_{join} becomes better. $E_{dynamic}$ combines two methods' advantages and its overall precision is the best. In Fig.3 (a) and (c), the KL error and MAPE error of $E_{dynamic}$ and E_{join} are smaller than $E_{selectivity}$ after processing 10 batches of data. Due to the computation complexity and statistical information maintaining, in Fig. 3 (b) and (d), applying E_{join} finishes earlier than applying $E_{selectivity}$. Similar to the precision results, the $E_{dynamic}$ has good efficiency results because it linear combines E_{join}'s estimation results.

Summary. Fig. 3 proofs that the $E_{dynamic}$ is better than existing result estimation strategies, it finishes earlier than $E_{selectivity}$ and has better accuracy than E_{join}. Please notice, $E_{dynamic}$ only computes $E_{selectivity}$ from the scratch at the first batch, so $E_{dynamic}$'s processing time is similar to E_{join}'s processing time.

6.3.2 Varying Method. We evaluate the quality of progressive results by the KL error and MAPE error of the progressive result distribution and the ground truth result distribution. We compare *QPJ* (in blue) with three baseline methods, *Prism* (in black), ConJ (in green), and Spark (in orange), in spatial join queries (Fig.4 (a) and (b)) and equi-join queries (Fig.4 (c) and (d)). The *Prism* baseline takes the same input progressive input as *QPJ* takes in each progressive computation round. To evaluate the ConJ, we divide the output results computed by ConJ that are ranked based on user preference score into progressive output results, each progressive output result has the same output rate as *QPJ* has. To evaluate Spark, we output the Spark join results into a single file and write in the disk. The progressive output results are extracted by the same strategy as ConJ used. We use Spark[4] to run the equi-join queries and Spark beast library[5] to run the spatial join queries.

Fig.4 (a) and (b) show Spatial join results of the four methods and Fig.4 (c) and (d) show the equi-join results. We report the KL error and MAPE error of the progressive result in every round. The y-axis represents the errors and the x-axis represents the result rates. In Fig.4 (a) and (b), the spatial join experiments produce 10 progressive results and the last progressive result is the whole join results. The spatial join experiments partition the data into 6 partitions and batch it into 10 rounds. In spatial join experiments, the progressive results computed by *QPJ* always have the smallest KL and MAPE error than other methods computed. In Fig.4 (c) and (d), the equi-join experiments produce 30 progressive results. The equi-join experiments partition the data into 10 partitions and batch it into 30 rounds. When the results rates are smaller than 50%, the progressive results computed by *QPJ* always have the smallest KL and MAPE error than other methods computed.

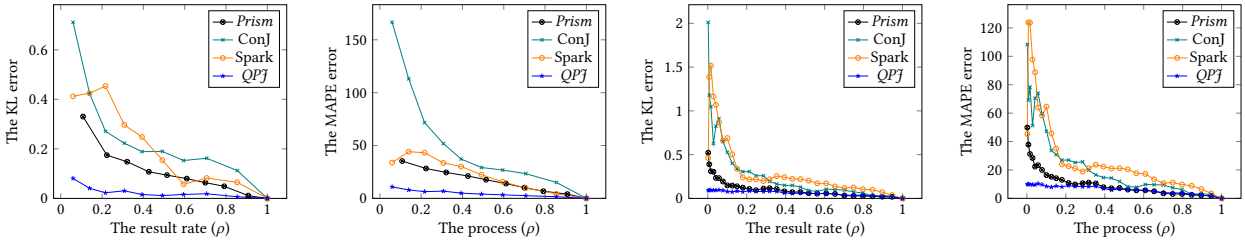
Summary. We prove the result distribution of progressive results computed *QPJ* are closest to the ground truth results than other methods, which means *QPJ* can produce progressive results to represent the ground truth results best. Therefore, in progressive query processing, *QPJ* can help users monitor the query output best and make the right decision earlier than other methods.

6.3.3 Varying Error Bound ε. We study the effects of the error bound ε from result quality and output rate perspectives. The result precision results are in Fig.5 (a) and (d). The output rate results are in Fig.5 (b) and (c). The spatial join experiments partition the data into 6 partitions and batch it into 10 rounds. The equi-join experiments partition the data into 10 partitions and batch it into 30 rounds. Since *Prism* produces progressive results that have better quality than ConJ and Spark, we only include *Prism* in the varying error bound experiments. Because there is no error bound control in baseline method *Prism*, the spatial join results of *Prism* in Fig.5 (a), (b), and (c) are the same.

The effects of the error bound ε: In both spatial join experiments and equi-join experiments (Fig.5 (a) and (d)), *QPJ* in blue always has smaller error than *Prism* in black and its output rate is also always smaller than *Prism*. In spatial join experiments, the accuracy gaps between the blue line and the black line are significant, which represents *QPJ* can return better results than *Prism*. In equi-join experiments, the precision gaps between *QPJ* and *Prism* are obviously in the beginning 40% of the output (the output rate ρ <0.4). With larger error bound, the precision of *QPJ* will decrease. Even increasing the error bound ε from 0.1 to 0.3, the precision gaps between *QPJ* and *Prism* are still significant in the beginning 40% (output rate ρ < 0.4). In summary, the result quality of *QPJ* is better than the quality of *Prism*.

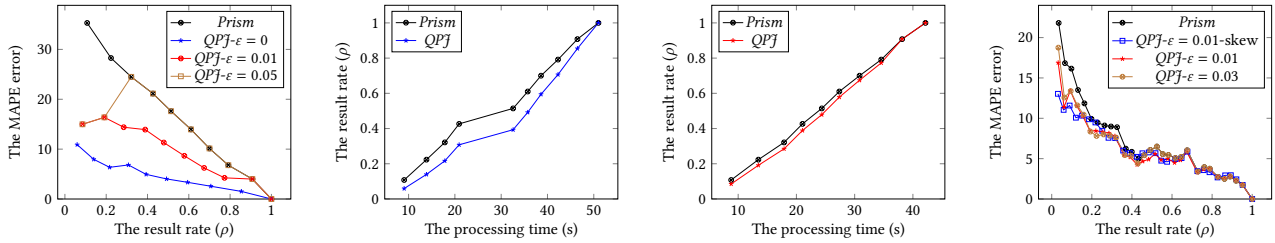
The effects of the error bound ε to output rate. *QPJ* can produce better results because it holds fewer results in memory. Therefore, compared with the baseline method, the result output rate of *QPJ* will be smaller than *Prism*. In Fig.5 (b) and (c), we report the sacrifices of output rate in *QPJ*. As shown in the results, *QPJ* does not lost too much output rate compared with the baseline method *Prism*. We propose the boost algorithm in Sec.5 Alg.1 to boost the result rate of *QPJ*. In Fig.5 (a) and (d), we can see that with larger error bound ε, the output rate difference is not significant between *QPJ* and *Prism*. We also compare with different error bound

[4]https://spark.apache.org/docs/latest/rdd-programming-guide.html#overview
[5]https://bitbucket.org/bdlabucr/beast/src/master/



(a) Spatial join, KL Error (b) Spatial join, MAPE Error (c) Equi-join, KL Error (d) Equi-join, MAPE Error

Figure 4: Compare the quality of progressive results of *QPJ* and existing frameworks.



(a) Spatial Join, MAPE Error (b) Spatial Join, $\varepsilon = 0$, result rate (c) Spatial Join, $\varepsilon = 0.01$, result rate (d) Equi-join, MAPE Error

Figure 5: The effects of the varying error bound.

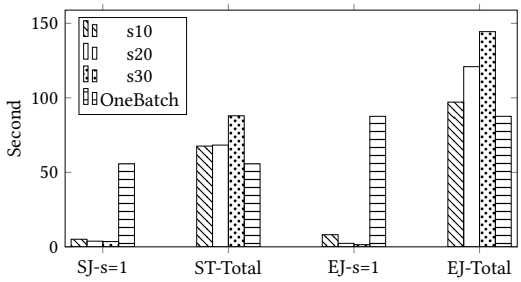


Figure 6: The comparisons of query responding time with different progressive computation rounds

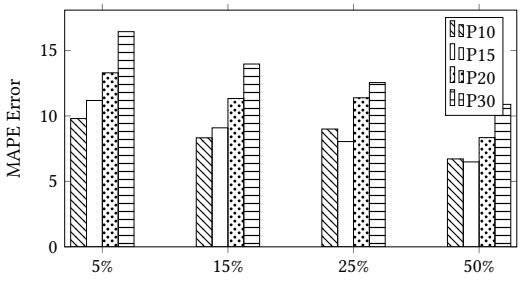


Figure 7: The output rate and accuracy comparisons with different numbers of partitions

ε, the changing of query result precision with more and more output results. With a larger error bound, the result quality becomes worse.

Summary. Combine the results shown in Fig.5, the input&output quality control of *QPJ* is better than single-choice output control of *Prism*. *QPJ* does not sacrifice too much output rate but can bring much better results than existing methods.

6.3.4 Analysis the effects of Data Distribution. We find an interesting fact that data distributions affect the precision of the frameworks. We test with spatial join with skew data distribution and equi-join with less data distribution. In the experiments of Fig.5, we shuffled both spatial datasets by chi-Square distribution. In experiments Fig.5 (d), we shuffled one of the Twitter datasets by chi-Square distribution and the other one by the discrete uniform distribution. The error gaps are more significant in the experiments tested with more skew distribution datasets. The figure shows that *QPJ* worked better when the datasets were more skew.

6.3.5 Varying S. In Fig.6, we study the effects of total computation round by comparing the *OneBatch* in black with *QPJ* on different progressive rounds: 10 rounds in green, 20 rounds in blue, and 30 rounds in orange. *QPJ* returns the results earlier than *OneBatch*. *QPJ* returns the earliest result in less than 5 seconds, while *OneBatch* needs more than 50 seconds for spatial join and more than 80 seconds for equi-join. When the total progressive computation is ten rounds $s = 10$, *QPJ* finishes the join queries sooner than s=20 and s=30. However, with more computation rounds, each round would produce fewer records and return the earlier results faster. 30 rounds experiment takes the shortest responding time to return the first results.

Summary. Progressive solutions are more suitable than non-progressive solutions in data exploration systems. The experiment with more computation rounds can refresh output results faster than with fewer computation rounds. However, due to the trade-off between total responding time and earlier results, extra computation and communication overhead increased with more computation rounds, which slowed down the overall query responding time.

6.3.6 Varying the number of partitions. We compare the quality of progressive results with the different numbers of partitions in Fig.7. The equi-join experiments adopt the hash partitioning on the joint key. We vary the number of hash buckets by changing the number of partitions and test with 10 partitions, 15 partitions, 20 partitions, and 30 partitions. In Fig.7, the y-axis is MAPE errors, the x-axis represents the output rate of each progressive result. We

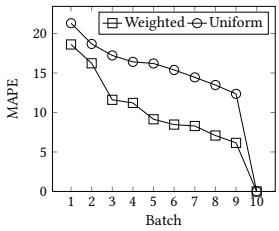


Figure 8: The comparisons of different sampling strategies.

report the MAPE error of the output rate reaches 5%, 15%, 25%, and 50%. 10 partitions experiment has the smallest error the output rate reaches 25%, and smaller number partition experiments can produce progressive results with smaller quality error.

Summary. When the number of partitions is reduced, the accuracy of progressive results is improved. The quality control component of *QPJ* works well with fewer partitions. However, when considering the distributed setting, the number of partitions affects the parallel execution. With more partitions, each worker node processes fewer data and would return results faster, which improves the performance. Besides, with more partitions, *QPJ* can have finer control over the quality of the progressive results.

6.3.7 Compare different sampling methods to select progressive output results. In Fig. 8, we compare the result distribution of progressive output results with ground truth result distribution. We test with equi-join with 10 coarser-level partitions, and each coarser-level partition contains 10 finer-level partitions. The weighted sampling method selects results that have smaller MAPE errors than results selected by the uniform sampling method. We show the weighted sampling method enables *QPJ* to have finer control over the quality of progressive output results.

7 CONCLUSION AND FUTURE WORK

We formally defined the progressive join problem to include the quality constraints in this work and propose an input&output control framework named *QPJ*. There are four improvements that can be considered. First, we will deploy *QPJ* on a distributed platform. Second, if the estimated result size is smaller than the previous round, current *QPJ* does not support hiding the existing outputted results. Third, we will extend *QPJ* to support the weighted sampling to pick spatial join output results. Fourth, we will extend the validation of *QPJ* to test with more complex queries, e.g. multi-way join, more types of partition functions, and the scalability.

REFERENCES

[1] Ning An et al. 2001. Selectivity estimation for spatial joins. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 368–375.
[2] Lars Arge et al. 1998. Scalable sweeping-based spatial join. In *VLDB*, Vol. 98. Citeseer, 570–581.
[3] et al. C. Zhu. 2014. Optimization of monotonic linear progressive queries based on dynamic materialized views. *Comput. J.* 57, 5 (2014), 708–730.
[4] et al. C. Zhu. 2016. Optimization of generic progressive queries based on dependency analysis and materialized views. *Information Systems Frontiers* 18 (2016), 205–231.
[5] Tsz Nam Chan et al. 2022. LIBKDV: a versatile kernel density visualization library for geospatial analytics. *PVLDB* 15, 12 (2022), 3606–3609.
[6] Bolin Ding et al. 2016. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD*. 679–694.
[7] Liming Dong et al. 2020. Marviq: Quality-Aware Geospatial Visualization of Range-Selection Queries Using Materialization. In *SIGMOD*. 67–82.
[8] Ahmed Eldawy and Mohamed F. Mokbel. 2019. All water areas in the world from OpenStreetMap. This includes coastal lines, lakes, rivers, pools, and others. https://doi.org/10.6086/N1668B70 Retrieved from UCR-STAR $https://star.cs.ucr.edu/?OSM2015/lakes&d$.
[9] Ahmed Eldawy and Mohamed F. Mokbel. 2019. Boundaries of parks and green areas from all over the world as extracted from OpenStreetMap. https://doi.org/10.6086/N1RX994T Retrieved from UCR-STAR $https://star.cs.ucr.edu/?OSM2015/parks&d$.
[10] Afrati Foto N et al. 2012. Fuzzy joins using mapreduce. In *ICDE*. IEEE, 498–509.
[11] Chandramouli Badrish et al. 2013. Scalable progressive analytics on big data in the cloud. *PVLDB* 6, 14 (2013), 1726–1737.
[12] Ding Mengsu et al. 2021. Progressive Join Algorithms Considering User Preference.. In *CIDR*.
[13] Jo Jaemin et al. 2019. Proreveal: Progressive visual analytics with safeguards. *TVCG* 27, 7 (2019), 3109–3122.
[14] Moritz Dominik et al. 2017. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *CHI*. 2904–2915.
[15] Procopio Marianne et al. 2019. Selective wander join: Fast progressive visualizations for data joins. In *Informatics*, Vol. 6. MDPI, 14.
[16] Qian Lin et al. 2015. Scalable distributed stream join processing. In *SIGMOD*. 811–825.
[17] Sameer Agarwal et al. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.
[18] Wee Hyong Tok et al. 2008. A stratified approach to progressive approximate joins. In *EDBT*. 582–593.
[19] Wongsuphasawat Kanit et al. 2017. Voyager 2: Augmenting visual analysis with partial view specifications. In *CHI*. 2648–2659.
[20] Wang Zhe et al. 2021. Neuralcubes: Deep representations for visual data exploration. In *BigData*. IEEE, 550–561.
[21] Yang Jianye et al. 2018. Efficient set containment join. *VLDBJ* 27, 4 (2018), 471–495.
[22] Yu Jia et al. 2020. Tabula in action: a sampling middleware for interactive geospatial visualization dashboards. *PVLDB* 13, 12 (2020), 2925–2928.
[23] Zhao Zhuoyue et al. 2018. Random sampling over joins revisited. In *Proceedings of the 2018 International Conference on Management of Data*. 1525–1539.
[24] Zhao Zhuoyue et al. 2020. Efficient join synopsis maintenance for data warehouse. In *SIGMOD*. 2027–2042.
[25] Hector Garcia-Molina. 2008. *Database systems: the complete book*. Pearson Education India.
[26] Peter J Haas, Jeffrey F Naughton, S Seshadri, and Arun N Swami. 1996. Selectivity and cost estimation for joins based on random sampling. *J. Comput. System Sci.* 52, 3 (1996), 550–569.
[27] Peter J Haas and Arun N Swami. 1995. Sampling-based selectivity estimation for joins using augmented frequent value statistics. In *ICDE*. IEEE, 522–531.
[28] Dawei Huang et al. 2019. Joins on samples: A theoretical guide for practitioners. *arXiv preprint arXiv:1912.03443* (2019).
[29] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1 (2007), 7–es.
[30] Jianfeng Jia et al. 2016. Towards interactive analytics and visualization on one billion tweets. In *SIGSPATIAL*. 1–4.
[31] Yuyu Luo et al. 2020. Visclean: Interactive cleaning for progressive visualization. *PVLDB* 13, 12 (2020), 2821–2824.
[32] Jignesh M Patel and David J DeWitt. 1996. Partition based spatial-merge join. *ACM Sigmod Record* 25, 2 (1996), 259–270.
[33] Johns Paul et al. 2020. Poet: an Interactive Spatial Query Processing System in Grab. In *SIGSPATIAL*. 477–486.
[34] Marianne Procopio et al. 2021. Impact of cognitive biases on progressive visualization. *TVCG* 28, 9 (2021), 3093–3112.
[35] Rahman Sajjadur et al. 2017. I've seen" enough" incrementally improving visualizations to support rapid decision making. *PVLDB* 10, 11 (2017), 1262–1273.
[36] Salman Ahmed Shaikh et al. 2020. GeoFlink: A Distributed and Scalable Framework for the Real-time Processing of Spatial Streams. In *CIKM*.
[37] Yufei Tao. 2022. Algorithmic Techniques for Independent Query Sampling. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 129–138.
[38] Wee Hyong Tok and Stéphane Bressan. 2013. Progressive and approximate join algorithms on data streams. In *Advanced query processing*. Springer.
[39] Wee Hyong Tok, Stéphane Bressan, and Mong Li Lee. 2006. Progressive spatial join. In *SSDBM*. IEEE, 353–358.
[40] Twitter. 2019. Twitter Data. https://twitter.com/?lang=en
[41] Wikipedia. 2022. Chi-square Distribution. $https://en.wikipedia.org/wiki/Chi-squared_distribution$
[42] Wikipedia. 2022. Discrete Uniform Distribution. $https://en.wikipedia.org/wiki/Discrete_uniform_distribution$
[43] Dong Xie, Jeff M Phillips, Michael Matheny, and Feifei Li. 2021. Spatial independent range sampling. In *Proceedings of the 2021 International Conference on Management of Data*. 2023–2035.
[44] Jia Yu and Mohamed Sarwat. 2021. GeoSparkViz: a cluster computing system for visualizing massive-scale geospatial data. *PVLDB* 30, 2 (2021), 237–258.