DeepOLA: Online Aggregation for Deeply Nested Queries

Nikhil Sheoran
sheoran2@illinois.edu
University of Illinois at Urbana-Champaign

ACM Reference Format:
Nikhil Sheoran. 2021. DeepOLA: Online Aggregation for Deeply Nested Queries. In *SIGMOD '22: Student Research Competition, 2022.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

With the advent of data-driven operating model, deriving useful insights from big data analysis has become all the more important. But the ever-increasing volume of data has made obtaining insights at "rates resonant with the pace of human thought" [4] more challenging. Online Aggregation (OLA) [3] is a technique that tries to counter this challenge by incrementally improving the query result estimates and allowing the user to observe the query progress as well as control its execution on the fly. More specifically, OLA provides the user with an approximate estimate of the query result as soon as it has processed a small portion (hereafter referred to as a *partition*) of the data. The user based on his latency-error trade-off requirements can choose to stop the execution of the query.

OLA was first proposed in [3] but it focused mainly on single-table queries and group-by aggregates. [1] further improved this by providing confidence intervals measures for the single table and multi-table aggregate queries with JOIN and selection predicates. Further work [2, 5] was done to optimize the query processing over multi-table joins including Ripple Join and its related optimizations. The work by [6] looks at online aggregation from the context of incremental maintenance of materialized OLAP cubes specifically from the perspective of non-distributive aggregate functions. [9] focuses specifically on non-monotonic queries and proposes a mini-batch execution model that partitions the intermediate output into deterministic and uncertain sets.

The current literature lacks a well-defined general framework that can be used to perform online-aggregation for complex and deeply nested queries. The majority of the current research rely on various assumptions on the type and nesting of the queries they can support. In this work, we provide a generalized framework called DeepOLA, that, defines a mechanism to perform online aggregate for deeply nested queries over multiple partitions of the input tables. The framework applies to queries with incremental updates on both a single-table as well as any subset of multiple tables.

Our preliminary results on TPC-H dataset and queries shows that DeepOLA can obtain estimates within reasonable error in a fraction of the original query's runtime. Not only that, for certain queries, we observed that incremental evaluation provides the accurate result computation faster than evaluating the query at once on the complete data (See Figure 2).

SIGMOD '22, 2022, Woodstock, NY

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

2 FRAMEWORK

In this section, we describe the overall framework for DeepOLA. We introduce the term **Incremental Dataframe** and **Incremental Operation** to define the notion of data and operations in the context of online aggregation and then describe how we parse and process an incoming query.

2.1 Incremental Dataframe

An Incremental dataframe is a structured data (i.e. relation) to which more data may be added. We define three types of incremental dataframes:

- Append (DA): The update is always appended to the existing data as new rows. An example of such a relation is the sales transaction data where only new records are added to the dataset.
- Merge (DM): The update is merged with the existing data based on a key of the dataframe. This merging operation can lead to addition of new keys as well as updates in the values of the dataframe for the existing keys. An example of such a relation is aggregated daily sales where the addition of new sales transactions modifies the computed average for the sales record.
- Complete (DC): The data is already complete and is not updated in the context of query processing for online aggregation. An example of such a relation is a relation containing the zip code and the corresponding city and country information.

2.2 Incremental Operations

The incremental operations are an extended version of regular relational algebra operations, with the added property of the type of input and output incremental dataframes. Table 1 shows the input and output dataframe types for some of these operations.

Operation	Input	Output
Filtering (Π)	{DA, DM, DC}	{DA, DM, DC}
Projection (σ)	{DA, DM, DC}	{DA, DM, DC}
JOIN or UNION	DA, {DA, DM, DC}	{DA, DM, DA}
JOIN or UNION	DM, {DM, DC}	{DM, DM}
JOIN or UNION	DC, DC	DC
GROUP BY AGG	{DA,DM,DC}	{DM,DM,DC}

Table 1: Type of Input and Output Dataframes for different Incremental Operations

We define a `merge` function for each of the operations operating on DA type of incremental dataframe. The merge function merges

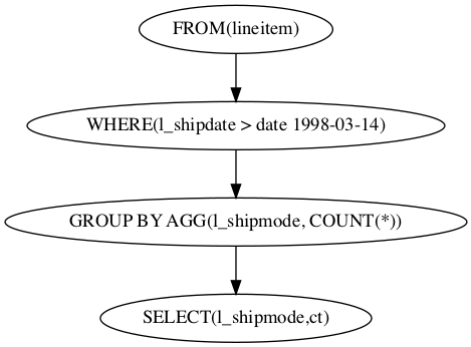

Figure 1: DAG for the example query in Section 2.3

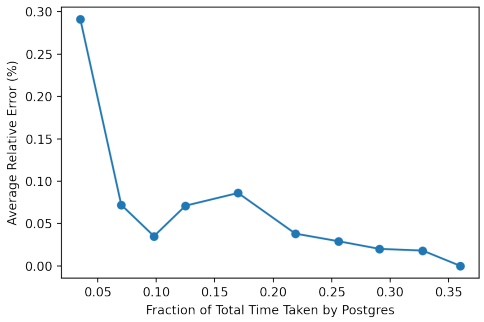

Figure 2: Error vs Latency Plot for a TPC-H Query processed with DeepOLA with 10 partitions. Each point corresponds to a processed partition.

the output of the relation on the current table with the output on the appends to that table while using an optional auxiliary information store c. Mathematically, for an operation R on the input dataframe I with appends ΔI, we define the merge operation g_R such that:

$$R(I \cup \Delta I) = g_R(R(I), R(\Delta I), c) \quad (1)$$

The key idea is that we want to *re-use the results* already computed on the dataframe to evaluate the result on the updated dataframe. An example of merge operation for $R = \Pi$ is: $\Pi(I \cup \Delta I) = \Pi(I) \cup \Pi(\Delta I)$.

2.3 Parsing a Query

We assume that a parser parses the input SQL query Q and provides us with a Directed Acyclic Graph $\mathcal{G}$ such that each *node* represents a relational operation with the associated arguments and each *edge* from a source to a destination node represents that the output of the source node is an input to the destination node. For example, for Q = `SELECT l_shipmode, COUNT(*) FROM lineitem WHERE l_shipdate > date 1998-03-14 GROUP BY l_shipmode`, Figure 1 shows the DAG structure obtained. The root nodes in the DAG (`FROM`) correspond to the input tables. The leaf node's (`SELECT`) output corresponds to the query output. Note that based on the input table's type (DA/DM/DC) and the operations defined in Table 1, we populate the type for each of the node in the graph $\mathcal{G}$.

2.4 Processing a Query

In this section, we describe the algorithm to perform online aggregation for an input DAG $\mathcal{G}$. First, we define a node n in DAG $\mathcal{G}$ to be **merge-able** if no nodes in the paths from the root nodes (inclusive) to that node (exclusive) are of type DM. Consider the DAG in Figure 1 with the table `lineitem` of type DA. Here, all nodes till `GROUP BY AGG` are merge-able whereas the node `SELECT` is not merge-able. Hence, when we process the query on incremental data, the partially computed results need not be merged at the `WHERE` node and can be merged at the `GROUP BY AGG` node (the last merge-able node in the path) before evaluating the `SELECT` node. Owing to this, we can delay the merge operation (which increases the size of the dataframe as it merges it with the current result at that node) to later nodes in a path in the DAG thus reducing the required re-computation - processing steps from the merged node onwards to the leaf node.

To process a DAG $\mathcal{G}$ on an input table I with k partitions such that $I = \cup\{I_1, I_2, ..., I_k\}$, we start with the update to the leaf nodes as I_i and propagate the update from that node to its children nodes. If any of the children nodes is of type DM, we merge the result at that node, store this merged result and then re-evaluate the nodes beyond this on the merged data. *Note that,* we only need to store the partial results for the first node of type DM (or the final output node) in any path. For nodes before that (of type DA), we can simply evaluate the output appends based on the input appends and for any nodes after the DM node, we need to re-evaluate the operations on the merged data.

3 EXPERIMENTS

We describe some preliminary results from our evaluation on the TPC-H dataset. We generated a partitioned dataset with a scale factor of 10 and 10 number of partitions using the tpch-dbgen[8] repo. The generated data was loaded in a PostgreSQL v14.1 database with the necessary indexing optimizations. The original query results along with the latency time were obtained by evaluating the queries against this database. DeepOLA's implementation uses the Python implementation of Polars [7] as the core data processing library. All the experiments were performed on an Intel Core i7 with 16 GB of memory.

Figure 2 describes the fraction of complete query evaluation's latency on the x-axis vs the estimation error on the y-axis averaged over 5 different runs for a representative TPC-H query[1]. As we can observe, our method not only provides a significant advantage for estimating the result with reasonable error, but also provides faster overall result computation - a 2.85x speed-up for the query in Figure 2.

As the computations are incremental, the time taken to read the next partition can be further reduced from the critical evaluation path by performing pre-fetch on the next partition to be read. We are performing further experiments with these optimizations and with various different queries and datasets to further strengthen our results.

[1]SQL Query: https://github.com/dragansah/tpch-dbgen/blob/master/tpch-queries/1.sql

REFERENCES

[1] P.J. Haas. 1997. Large-sample and deterministic confidence intervals for online aggregation. In *Proceedings. Ninth International Conference on Scientific and Statistical Database Management (Cat. No.97TB100150)*. 51–62. https://doi.org/10.1109/SSDM.1997.621151

[2] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data* (Philadelphia, Pennsylvania, USA) *(SIGMOD '99)*. Association for Computing Machinery, New York, NY, USA, 287–298. https://doi.org/10.1145/304182.304208

[3] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. 1997. Online Aggregation. *SIGMOD Rec.* 26, 2 (jun 1997), 171–182. https://doi.org/10.1145/253262.253291

[4] Z. Liu and J. Heer. 2014. The Effects of Interactive Latency on Exploratory Visual Analysis. *IEEE Transactions on Visualization and Computer Graphics* (2014).

[5] Gang Luo, Curt J. Ellmann, Peter J. Haas, and Jeffrey F. Naughton. 2002. A Scalable Hash Ripple Join Algorithm. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (Madison, Wisconsin) *(SIGMOD '02)*. Association for Computing Machinery, New York, NY, USA, 252–262. https://doi.org/10.1145/564691.564721

[6] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, and Hamid Pirahesh. 2002. Incremental maintenance for non-distributive aggregate functions. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 802–813.

[7] Pola-Rs. [n. d.]. Polars: Blazingly fast DataFrames in Rust Python. https://github.com/pola-rs/polars.

[8] Dragan Sahpaski. [n. d.]. TPCH-DBGEN. https://github.com/dragansah/tpch-dbgen.

[9] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-OLA: Generalized On-Line Aggregation for Interactive Analysis on Big Data *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 913–918. https://doi.org/10.1145/2723372.2735381