DEEPOLA: AN ONLINE AGGREGATION APPROACH TO APPROXIMATING DEEPLY NESTED QUERIES

BY

NIKHIL SHEORAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois Urbana-Champaign, 2022

Urbana, Illinois

Adviser:

Assistant Professor Yongjoo Park

ABSTRACT

For exploratory data analysis, it is often desirable to know what answers you are likely to get *before* actually obtaining those answers. This can potentially be achieved by designing systems to offer the estimates of a data operation result — say `op(data)` — earlier in the process based on partial data processing. Those estimates continuously refine as more data is processed and finally converge to the exact answer. Unfortunately, the existing techniques — called *Online Aggregation* (OLA) — are limited to a single operation; that is, we *cannot* obtain the estimates for `op(op(data))` or `op(...(op(data)))`. If this *Deep OLA* becomes possible, data analysts will be able to explore data more interactively using complex cascade operations.

In this work, we take a step toward *Deep OLA* with *evolving data frames* (*edf*), a novel data model to offer OLA for nested ops — `op(...(op(data)))` — by representing an evolving structured data (with converging estimates) that is *closed* under set operations. That is, `op(`*`edf`*`)` produces yet another *edf*; thus, we can freely apply successive operations to *edf* and obtain an OLA output for each op. We evaluate its viability with DeepOLA, an *edf*-based OLA system, by examining against state-of-the-art OLA and non-OLA systems. In our experiments on TPC-H dataset, DeepOLA produces its first estimates 12.8× faster (median) — with 1.3× median slowdown for exact answers — compared to conventional systems, while even producing faster exact results for some queries (i.e. no overhead). Besides its generality, DeepOLA is also 1.92× faster (median) than existing OLA systems in producing estimates of under 1% relative errors.

To my parents and sister, for their faith, love and support.

ACKNOWLEDGMENTS

I am very grateful to Professor Yongjoo Park for giving me an opportunity to work with him as part of the CreateLab, University of Illinois at Urbana-Champaign (UIUC). He has supported me, provided me with great guidance and advice throughout my journey at UIUC without which completing this thesis would not have been possible. I am also very thankful to Supawit Chockchowwat (CreateLab, UIUC) for his constant input, help with the implementations and various discussions throughout the project.

I would like to thank the eager and hard-working undergraduate students — Romero Risa, James Wei, Arav Chedda, Suwen Wang and Riya Verma who have helped me with various different parts throughout the project.

I am thankful to my sister and my parents for motivating me to constantly keep pushing. I would like to thank my friend and flatmate Nikunj for being the constant support throughout my time at UIUC. The regular discussions and nudges have been crucial to making this into a reality.

An early abstract of this thesis was published and presented at the Student Research Competition, Special Interest Group on Management of Data (SIGMOD) 2022 [1]. A more complete version has been submitted and is in review for SIGMOD 2023. This research used credits from the generous grant by Microsoft Azure.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Majority of the businesses are moving towards data-driven operational models, where deriving useful insights from big data becomes a key part of the operations. The increasing volume of the data has made it difficult to obtain insights at "rates resonant with the pace of human thought" [2]. To counter this, various query approximation methods are used. The goal of these methods is to be able to obtain "approximate but good enough" results that doesn't alter the insight obtained from the data while also significantly reducing the query processing time and computation costs.

Online Aggregation (OLA) [3] is one such method that provides continuously refining estimates of the results while providing the user with a visibility on the query progress. The user can choose to terminate the query execution early depending on their latency-error trade-off requirements. Unfortunately, the existing OLA has a common limitation, which makes it *not* the first choice for today's data exploration. That is, the existing OLA *precludes subsequent operations on previous OLA outputs.* To illustrate this, consider the example data analysis session[1] expressed in pandas-like methods [4] shown in Listing 1.1.

```
1 lineitem = read_csv('...')
2 # item count for each order
3 order_qty = lineitem.sum(qty, by=orderkey)
4 # select only the large orders
5 lg_orders = order_qty.filter(sum_qty > 300)
6 # find the customers with biggest order sizes
7 lg_order_cust = lg_orders.join(orders).join(customer)
8 top_cust = lg_order_cust.sum(sum_qty, by=name).limit(100)
```

Listing 1.1: An example data analysis session

The first output (L3) is aggregated/filtered again to find the top customers (L5-L8). Existing OLA can incrementally compute the first output (i.e., `order_qty`), but it cannot be subsequently processed for filter/join/sum in an OLA fashion until its final answer is obtained. Specifically, the existing OLA has two limitations. First, it treats every query independently without reasoning about how its output may be consumed by subsequent operations. Second, it cannot handle arbitrarily deep queries; that is, even if we compose a (long) query for directly computing `avg_order_size`, the aggregation over aggregation — with correct adjustments — cannot be produced (*note:* this problem is different from incremental view maintenance, which always produces the exact results).

In this work, we tackle this limitation with *evolving data frames* (or *edf*), a novel data/processing model designed to enable Deep Online Aggregation — the ability to apply subse-

[1] This example data analysis is a rewritten version of TPC-H query 18.

quent operations to previous OLA outputs for another OLA output. For each operation, *edf* offers *converging* estimates for the final answer — with diminishing expected errors — relying on a common assumption that unseen data mimics the observed; once the entire data is processed, each *edf* exactly matches the one that can be obtained by conventional data processing systems. To evaluate the viability of our approach, we implement DeepOLA, an *edf*-based OLA system, and examine its performance against existing OLA systems (ProgressiveDB [5], WanderJoin [6]) as well as modern data systems/libraries (Presto [7], PostgreSQL, Polars [8]).

To our knowledge, no previous work has formally studied a data model for deep online aggregation and has evaluated its efficiency for practical use cases with comprehensive experiments.

1.1 CHALLENGES

Given a query (or an operation), existing OLA can be understood as a process that converts an input data into a series of intermediate/final results, where notably, the input and the output are of different types, which is the fundamental reason that OLA cannot be applied to the results of OLA. Specifically, we observe the following challenges. First, the existing model for structured data (which we call *dataframe*) is insufficient for expressing progressively changing data frames which may contain approximate attribute values and their row counts may change. Second, the existing set-oriented (or relational) operations are designed for a final data frame, not an approximate one; simply applying regular operations to an evolving data frame may produce biased values because partial data must be regarded as a sample. Third, offering high-performance is critical. Any OLA-driven extensions to the existing data model may incur overhead, which must be small enough to still deliver significantly more interactive computing compared to conventional *all-at-once* approaches.

1.2 PROPOSED APPROACH

Our new data model, *edf*, is *closed* under a class of set operations; that is, an *edf* expresses an evolving OLA output, which when transformed by a set operation, again produces yet another *edf*. Specifically, *edf* has the following key characteristics. First, each *edf* always converges to the exact/final answer once the entire data is processed. Second, to ensure that an operation on an *edf* produces another *edf*, our set operations — expressed using `map`, `filter`, `join`, and `agg` — maintains two unique properties inside each *edf*, i.e., *mu-*

table attributes and *cardinality growth*, which are key to producing accurate estimates (section 3.2.4). Third, our internal processing is designed to minimize redundant computation whenever possible, which is essential for fast response.

1.3 CONTRIBUTIONS

This work shares the following findings:

1. Our new data model, *evolving data frames* (*edf*), enables successive OLA operations. (chapter 3)

2. Our processing model, representing common set operations, can transform an *edf* into another *edf* (with correct properties) relying on our internal inference technique. (chapter 3)

3. DeepOLA's multithreaded implementation for OLA can offer high processing speed with pipelined parallelism. (chapter 4)

4. DeepOLA can produce an intermediate result 12.8× faster than the associated final answer, while incurring (only) 1.3× overhead compared to non-OLA. The median relative accuracy of the first intermediate answer for TPC-H datasets is 2.70%, which converges quickly towards zero. (chapter 5)

The thesis is organized as follows. Chapter 2 discusses the prior work in the field of AQP and OLA. Chapter 3 then discusses the motivation behind the need of deep online aggregation, the proposed data model and how the different operations and internal processing works for this data model. Chapter 4 discusses the implementation details and then Chapter 5 discusses the various experiments performed to evaluate the performance of DeepOLA.

CHAPTER 2: BACKGROUND

In this chapter, we describe various prior work in the field of approximate query processing, more specifically online aggregation.

2.1 APPROXIMATE QUERY PROCESSING

Approximate Query Processing (AQP) allows users to interactively analyze large datasets at a fraction of the costs of executing exact queries. Despite the years of research in AQP, it has been less successful in supporting deeply nested queries [14].

Existing AQP methods includes various synopses-based techniques [15] using samples [13, 16, 17, 18], wavelets [19], histograms [20, 21], sketches [22], etc. These methods generate synopses based on the original data and use these synopses to answer any incoming queries. Since the size of synopses is usually smaller than the original data size, the computation cost associated with answering the queries is significantly reduced. Challenges with these methods include maintenance of synopses, requirement of a diverse set of synopses, additional cost associated with storage, the inability to provide exact results even if the user is willing to wait.

Recent ML-based works formulate AQP as a data learning problem — through density estimators [23, 24], regression models [24], generative models [25, 26, 27]. DeepDB [23] tries to learn a joint probability distribution over the data through a relational sum-product network. This formulation allows them to use the same learned distribution for multiple different tasks including cardinality estimation, ML-related tasks, query approximation, etc. DBEst [24] models query approximation as a learning problem - trying to learn a density estimator and a regression function. The key challenge arises with attributes involving large number of groups. VAE-AQP [27] uses a generative-model based approach to query approximation. They train a variational auto-encoder model and use it to generate samples at runtime, based on which queries are answered.

More recently, query-aware generative models have been used to improve approximation error for low-cardinality queries [28, 29]. Electra [29] uses a conditional generative model to generate samples based on the predicates specified in the queries. Such a conditional generative model can generate samples specific to the input query predicates, thereby improving performance on low-cardinality queries. Challenges with these approaches include the high-cost of model maintenance and re-training, the limited set of supported queries and difficulty in providing correctness bounds.

System/Method	OLA?	Deep?
OLA (Hellerstein) [3]	✔	✗
RippleJoin [9, 10]	✔	▲
WanderJoin [6]	✔	▲
G-OLA [11]	✔	▲
QuickR [12]	✗	✔
VerdictDB [13]	✗	▲
Ours (DeepOLA)	✔	✔

Table 2.1: Summary of existing work. ▲/✗ indicates limited/no support.

2.2 ONLINE AGGREGATION

Hellerstein et al. [3] first introduced the idea of OLA, that performs aggregation online in order to allow users to both observe the progress of their queries and to control its execution on the fly. The queries were initially limited to simple aggregate queries without support for join/nested subqueries. Various works [6, 9, 10, 30, 31] have built on top to improve the performance as well as increase the extent of queries supported.

RippleJoin [9] extended OLA for join queries. The algorithm proceeded by computing repeated joins of the newly selected rows from the data with the previously seen data. Owing to the repeated joins with the existing table, the complexity for joining multiple tables becomes exponential. [10] improves online join algorithms for queries with low cardinality or high number of groups. [30] provides OLA support for sort-based join algorithms. [31, 32] provides a scalable disk-based approach while providing better statistical bounds for sort-based join algorithms. WanderJoin [6] counters the exponential complexity by using a random-walk based approach along with indexes for joining multiple tables. Despite the years of research, the support for nested aggregates and sub-queries have been very limited in the literature.

G-OLA [11] generalizes OLA to nested predicates by dividing the processed tuples in certain and uncertain sets based on running estimates. Since its efficiency relies on quality of the estimates, incorrect estimation can lead to re-computation and a more relaxed slack leads to increased size of uncertain sets thereby impacting performance. QuickR [12] approximates complex ad-hoc queries by injecting samplers on the fly. The improvements are significant in cases there are multiple passes over data. Although the query approximation is not online, QuickR does support nested complex queries. Table 2.1 gives a quick summary of the extent of queries supported by different prior works.

2.3 TPC-H BENCHMARK

TPC-H [33] is a decision support benchmark that provides a data generation script as well as a set of 22 queries that simulate various different scenarios in decision support. The queries range from single table to multi-table join queries, with optional group-by, aggregates, nested sub-queries — thereby providing a representative set of data exploration analysis queries.

CHAPTER 3: DEEP ONLINE AGGREGATION

In this chapter, we first describe why we need a novel data model for deep online aggregation and the properties that it should satisfy. Then we propose a data model and provide details about how it can process different relational operations.

3.1 MOTIVATION

The outputs of an online aggregation computation needs to be represented as *types*. For example, integers (e.g., -1, 0, 1, 2) are closed under addition; thus, we can apply successive additions to the outputs of previous additions, e.g., (1 + 3) + 3 = 4 + 3, without being concerned about how the value 4 (= 1+3) is originally obtained. Likewise, database relations (representing 2-D structured data) are closed under relational operations (e.g., projection, aggregation, join).

In contrast, the existing OLA is designed to consume a relation as an input and outputs a series of relations, each representing a converging estimate for the final answer (with expectedly decreasing errors). Thus, the input to and the output from OLA are of different types (i.e., relations are not closed under OLA), which makes it non-trivial to apply successive OLA to the outputs of previous OLA.

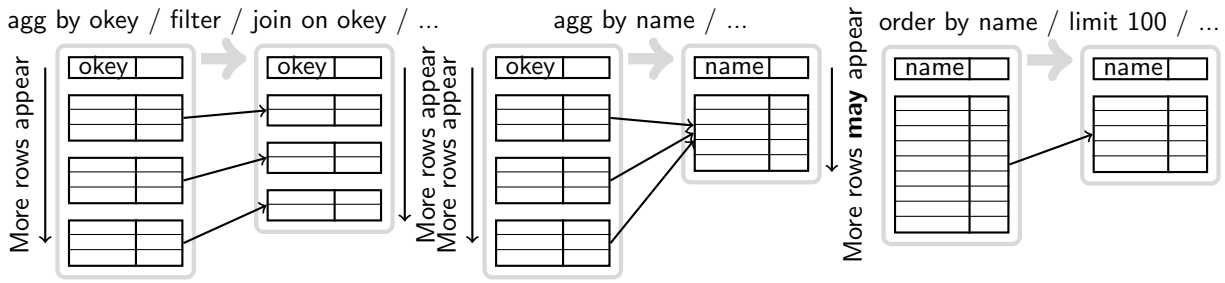
It might be possible to apply another OLA to each estimate relation; however, first, it is not straightforward how to interpret this, and second, the number of output estimates grows exponentially with the number of operations if we naïvely apply OLA to each estimate. This motivates us to introduce a new type that is closed under OLA, which we call evolving data frame (*edf*).

3.2 ANALYSIS OF OLA OPERATIONS

In this section, we discuss how different operations (e.g., agg, filter, join, limit) may alter an input data frame into another form, which serves as the basis of our data model in sections 3.3 and 3.4.

3.2.1 Order-Preserving Local Operations

Suppose an input data frame (e.g., `lineitem` table) — getting read from csv file(s) — contain the raw data sorted/clustered on a key (e.g., `orderkey`). In processing row-wise



(a) Order-preserving Local Op (b) Shuffle Op with Inference (c) Shuffle Op w/o Inference

Figure 3.1: Example data frame transformations for OLA. For each case, an input data frame (left) transforms to an output data frame (right) by one of the operations noted on top. The output cardinality may grow as processing more data.

filters and maps, newly appearing rows in an input data frame do not affect the results of already processed rows.

Specifically, let `df` be an input data frame consisting of two partitions, i.e., `df` = [$\texttt{df}_1$, $\texttt{df}_2$], where "[]" indicates union/append. For such a *local operation* `op`, `op(df) = [op(`$\texttt{df}_1$`), op(`$\texttt{df}_2$`)]`; thus, unlike other cases described shortly, computing `op(`$\texttt{df}_2$`)` is independent of $\texttt{df}_1$, which makes it possible to incrementally produce the output. Likewise, inner/left join (e.g., joining `lineitem` with `orders`) is also an order-preserving local operation since `join(dfa, dfb) = [join(`$\texttt{dfa}_1$`, dfb), join(`$\texttt{dfa}_2$`, dfb)]`; its physical plan may opt for different join algorithms such as progressive-merge [30] or hash joins, depending on right tables. For instance, the right table `dfb` may be converted to an in-memory hash table and repetitively used for $\texttt{dfa}_1$ and $\texttt{dfa}_2$.

3.2.2 Shuffling Operations with Inference

If an input data frame is aggregated by a non-key attribute, e.g., `lg_order_cust.sum(sum_qty, by=name)`, we need special considerations for three reasons. First, already produced output (raw) aggregate values may change as we process more data from an input. Second, raw aggregate values may need to be scaled appropriately to produce accurate — desirably unbiased — estimates. Third, more rows (containing new grouping key values) may appear in the output as we process more data from an input, which need to be modeled quantitatively for subsequent operations that will consume this output data frame.

Figure 3.1b depicts the data flows of this case: the newly appearing rows in the input data frame may affect an already produced output. Specifically, let `df = [`$\texttt{df}_1$`, `$\texttt{df}_2$`]`. For a *shuffling operation* `op`, `op(df) = op(`$\texttt{df}_1$`)` $\bigoplus$ `op(`$\texttt{df}_2$`)`, where $\bigoplus$ indicates a key-based

merge, which can be expressed as A $\bigoplus$ B = agg(union(A, B), by=key).[2]

The result of this merge must be scaled to produce accurate estimates if *more rows may appear in the input data frame* during OLA. For example, if the input data frame represents a base table for which more data are being retrieved, the currently observed part(s) must be considered as a sample of the input data; thus, the raw sum values need to be scaled up in consideration of the ratio between the current row count and the entire data size (which serves as a sampling ratio). In contrast, if the input data frame is, for example, a result of an aggregation (with a low-cardinality grouping attribute), we are unlikely to see (many) new rows in the input data frame; thus, the currently observed set is the entire set, thereby not requiring additional scaling. While the individual aggregate values may be approximate, since they are already converging estimates of the final answer, the raw sum values (without additional scaling) are also converging estimates of the output (section 3.4.3).

3.2.3 Shuffling Operations without Inference

Some operations, such as `order-by` and `limit`, must consume the entire input, for which no special treatment can be additionally applied to improve the quality of an output. In these cases, upon a change of an input, the output simply needs to be recomputed in its entirety, which, unlike Cases 1 and 2, cause inevitably redundant computation. For large-scale aggregation, however, these Case 3 operations typically appear in latter stages to limit/sort the result for user consumption (e.g., bar charts); thus, the overhead incurred by such redundant computations is insignificant in the context of overall computations. Nevertheless, if a user intention is, for example, to sort the entire data and to persist its result on disk, OLA frameworks (including ours) do not offer additional benefits.

3.2.4 Required Properties

The case analysis above reveals two types of changes that may appear in a data frame: (1) changes to attribute values (e.g., as aggregating more input rows) and (2) cardinality growth (e.g., as filtering input rows as they appear).

- Mutable Attributes: Let a *mutable attribute* be an attribute whose values may change as more data is processed whereas a *constant attribute* be an attribute whose values

[2]Merge operations are applicable to sum-like (or *mergeable*) operations, for which *addition* can be defined. Accordingly, avg() needs to computed by separately computing sum() and count(). One notably hard case is count-distinct, for which we maintain exact sets (not HLL-based sketches [34]).

never change. It is useful to distinguish mutable attributes from constant attributes because the input attribute types affect how we should (re-)compute the output.

For example, filtering on a constant attribute (e.g., `name like '%east%'`) can be processed incrementally (Case 1) whereas filtering on a mutable attribute (e.g., `sum_qty` $>$ `200`) requires re-computation (Case 3).

- Cardinality Growth: As observed in Case 2, how many rows are likely to appear in an input data frame must be captured to properly estimate output aggregates. To this end, we define *cardinality growth*: the relationship between query progress and group cardinality (i.e. the number of rows belonging to an aggregate group).

  After studying a diverse family of cardinality growths, we can select the most fitting growth to predict the final aggregates. For example, suppose we know that the group cardinality grows linearly with the query progress; then, if the query progress is at 25%, we would expect to see 4× rows in the final group cardinality.

3.3 DATA MODEL

This section describes evolving data frames' (*edf*) data model, the set of operations supported on it and the current limitations.

3.3.1 Evolving Data Frames (EDF(s))

An evolving data frame (*edf*) represents a progressively changing structured data (i.e, data frame) — with new rows appearing and/or changing attribute values — with the formal definition:

```
edf  := list((attr1, attr2, ..., attrM))
attr := constant_attr | mutable_attr
```

Listing 3.1: Formal definition of evolving data frames

where `(attr1, attr2, ..., attrM)` defines a schema; one or more (`constant`) attributes serve as the *primary key* (or simply *key*) to uniquely identify tuples. An *edf*'s row count (i.e., the length of a list) may increase over time, and the values for `mutable_attr` may change.

An *edf*'s list is organized using one or more *partitions*, where each partition is (simply) a subset of the list stored/accessed together (e.g., on a storage device). An *edf* can optionally have a *clustering key*, a list of attributes determining the placements of rows among partitions; for example, if an *edf*'s clustering key is `orderkey`, say a partition includes the rows

with `orderkey` between 1 and 10; then, other partitions must not contain the rows with `orderkey` between 1 and 10. A clustering key may also be present for the *edf*(s) created as results of operations on other *edf*(s), as we describe in section 3.4.

3.3.2 Creating EDF(s)

There are two ways to create *edf*(s). An *edf* can be created directly from a data source or an *edf* can be created as a result of operation on another *edf*, as follows:

```
read := data_source -> edf
edf_op := (edf, op) -> edf
op  := agg(attrs, by) | filter(predicate)
     | map(function) | join(df, options)
agg := sum | count | avg | count_distinct | min | max
     | var | stddev
```

Listing 3.2: Creating and performing operations on evolving dataframes

where details of individual operations are described in section 3.3.4. When creating an *edf* from a data source, a clustering key is obtained from metadata. These operations and types of aggregations are sufficient to express all 22 TPC-H benchmark queries [33].

3.3.3 Accessing EDF Values

To represent an evolving data frame, an *edf* maintains *states*. The latest state can be obtained with ***edf*`.get()`**, where each state expresses a converging estimate of the final answer with the latest state being the most accurate in expectation. For example, if an *edf* is for `lineitem.count(by=linestatus)`, the count value in each row of the *edf* is an unbiased estimate of the final count value for the same group (i.e., they are equal in expectation). The latest state is obtainable via ***edf*`.get()`**. If ***edf*`.is_final`** is true, the latest state holds the final answer; the `is_final` flag is set by the system as soon as the system finds there will be no more data to process (when receives an `EOF` signal). The final answer can be obtained by ***edf*`.get_final()`**, which may block until processing the entire data (if not already processed).

3.3.4 User Operations on EDF(s)

Our system (DeepOLA) implements relational operations such as projection (map), join, selection (filter), and aggregation, in a unique manner — to maximize OLA opportunities — as follows.

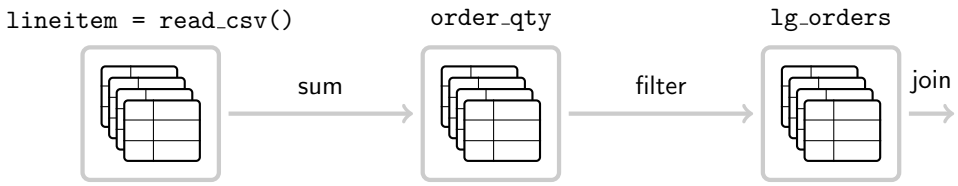


Figure 3.2: User-view of evolving data frames (*edf*) and operations on them.

- Map: *edf*`.map()` resembles projection operations such as selecting a subset of attributes, creating derived attributes, etc. What is unique to our `map()` is that the function in its argument is applied to one or more *partitions* instead of each row. Specifically, let *edf* `= [p1, p2, ..., pK]` where `pi` is a partition of rows; then, *edf*`.map(func)` creates another *edf*2 such that *edf*`2 = [func([p1, p2]), func([p3, p4]), ..., func([pK-1, pK])]`. That is, `func` maps a data frame to another data frame where each input data frame is a set of partitions. Here, the number of partitions passed to each `func` invocation — two in this example — is determined based on partition sizes. There are two reasons behind this design. First, this approach enables partition-specific (local) operations that are less trivial to express, e.g., finding two most ordered items within each order where an order consists of multiple (item, quantity) tuples. Expressing this using relational operations (in SQL) can involve less commonly used functions (e.g., `group_concat` [35], `find_in_set` [36]). Second, the approach easily enables efficient parallel processing without additional logic for parallelizing/vectorizing row-wise functions.

- Join: *edf*`.join(`*edf*`2, options)` joins *edf* with *edf*2 as specified in its `options`, e.g., method (inner/left) and join keys. Depending on the join keys and clustering keys, DeepOLA uses a different join method (i.e., hash or progressive-merge [30]). Specifically, if both *edf* and *edf*2 are clustered on their respective join keys, DeepOLA performs a merge join; otherwise, DeepOLA performs a hash join with *edf* as the *probe table* and *edf*2 as the *build table* (used for creating a hash table). If multiple joins are chained (e.g., *edf*`.join(`*edf*`2).join(`*edf*`3)`) and hash joins must be used, DeepOLA effectively performs the right-deep join by constructing hash tables in parallel for *edf*2 and *edf*3, which is effective for star schema models [37].

- Aggregate: *edf*`.agg(cols, by_attr)` aggregates a group of rows (for each `by_attr`) where `agg` is one of allowed aggregate functions. For Deep OLA, we treat aggregation specially because to generate accurate/unbiased estimates, the results of partial aggregation may need adjustments in consideration of the ratio between an observed

data frame size and the full data frame size, while the full data frame size may also be uncertain if, for example, the data we are aggregating is a result of another aggregation, thereby requiring further inference. sections 3.4 and 3.5 describe more on our inference logic.

- Filter: *`edf`*`.filter(predicate)` resembles the selection operation in relational algebra (or the `where` clause in SQL); that is, the operation produces another *edf* consisting only of the rows satisfying the supplied predicate. Like *`edf`*`.map(...)`, the predicate is applied to one or more *partitions* together. In general, `filter()` can be understood an alias of `map()` that may produce an empty set as an output. Specifically, for *`edf`* `2 = [func([p1, p2]), func([p3, p4]), ..., func([pK-1, pK])]`, any of `func` may produce an empty set.

We have described the four operations (i.e., map, join, aggregate, filter) from a user's perspective; however, the internal processing may differ based on schemas/operations, which we describe in section 3.4.

3.3.5 Limitations

There are cases where some operations must block (e.g., filtering/joining on mutable attributes) to produce correct results while minimizing redundant computations. While our internal processing logic (section 3.4) can distinguish such cases, it may be less straightforward to end users especially when they are new to our framework. To maximally exploit deep OLA opportunities, more advanced users may carefully organize data and operations, which may be considered as *skill* (like providing join hints in RDBMS); however, one may argue that this means the system is not intelligent enough to automatically optimize user operations. The scope of this work is to construct the foundational building blocks for Deep OLA without optimizing an end-to-end declarative query as performed by RDBMS with cost-based optimizers, which we leave as future work.

3.4 INTERNAL PROCESSING

In addition to the schema described in section 3.3.1, each *edf* maintains two additional properties, namely *progress* and *growth*, to characterize its evolution quantitatively.

- Progress: Progress ($0 \leq t \leq 1$) is the ratio between the number of (original) *input* tuples that have been read/processed thus far and the *total* number of the (input)

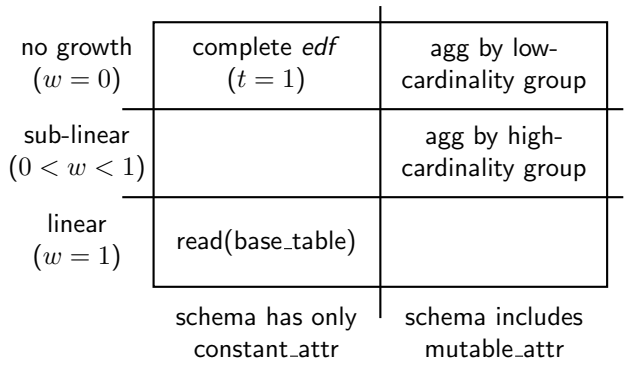


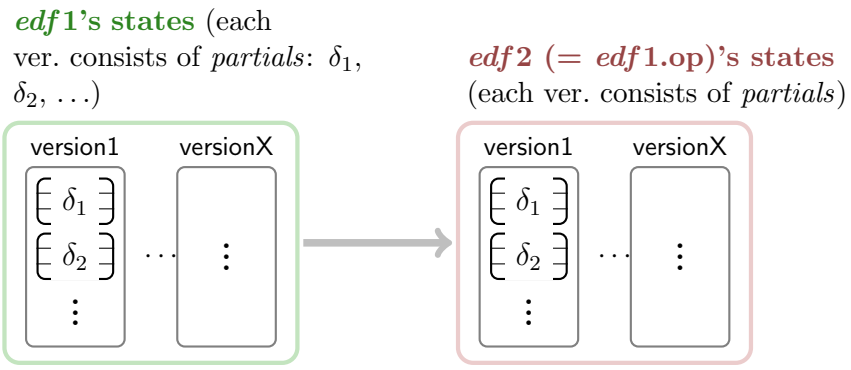
Figure 3.3: *edf* types categorized by degree of growth w on Y-axis and attribute types on X-axis with examples in boxes.

tuples that must be processed to obtain the final answer; the total tuple count comes from metadata. For example, if a base table consists of ten equal-sized partitions, and we have read/processed only one of them, t is 1/10 (= 0.1). On the other hand, if the entire data (e.g., ten out of ten partitions) are read/processed, t is 1. If t is 1, *edf*`.is_final=True`.

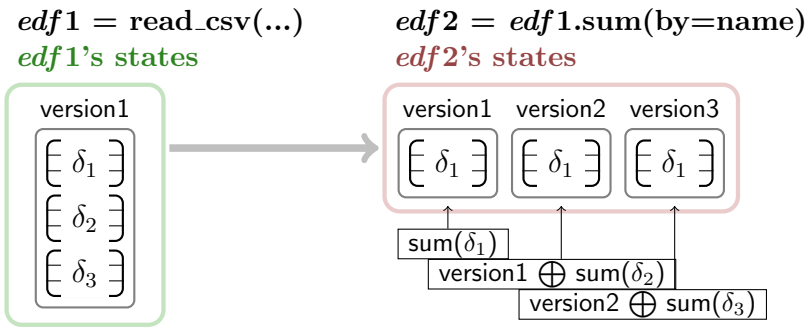
- Growth: Growth describes the growth of current tuple count to forecast the final tuple count. DeepOLA compactly models the growth as a monomial ct^w using past observations. fig. 3.3 gives some examples with different w values. Growth captures local tuple count, while progress t captures the query input ratio (between the current and the future). For instance, if we are computing an average (without grouping attributes), the output tuple count will always be one (unless empty); thus, $w = 0$ and $c = 1$. On the other hand, $t < 1$ if we are still reading/processing input data.

These variables — (c, w) and t — are more closely related if, for example, an *edf* represents a base table; then, w is equal to 1 and c is equal to the input size, because in this case, the output of this *edf* (or the data this *edf* represents) exactly matches the amount of input data retrieved from a data source (e.g., csv files in a directory). In other cases, however, w may be less than 1, suggesting sublinear growth. Notably, t may also be 1 (or close to 1) even before processing the entire data; For instance, if an *edf* represents the result of aggregation with log-cardinality grouping attributes — `lineitem.count(by=linestatus)` — the number of output rows is less likely to increase (while its aggregate values may change); thus, we have $t = 1$. Thus, $t = 1$ does not imply `mdf.is_final=True`; it simply indicates we are unlikely to see new grouping keys.

Figure 3.3 classifies the types of *edf* properties based on degree of growth (w) and attribute types (constant/mutable). Its cells list a few examples that would result in *edf*s with such



(a) General: An OP incrementally constructs states for a new *edf*



(b) Example: `sum` creates multiple versions, each having one partial

Figure 3.4: States-based representation of *edf* and operations on them (i.e., from *edf*1's extrinsic states to *edf*2's intrinsic states). Each *edf* (conceptually) defines a new state each time a partial is added to the latest version or a new version is created. An operation on *edf* creates a new *edf* (and states for it) with minimal computational redundancy.

properties. For example, if `edf = read(base_table)`, its schema consists only of constant attributes and its output size grows linearly with input data ($w = 1$). Another example is an *edf* representing the result of aggregation with high-cardinality grouping attributes (`students.count(by=first_name)`). If so, attribute values may change, and also, new grouping keys may appear (each time a new first name is encountered). Accordingly, its schema includes mutable attributes, and w is between 0 (no growth) and 1 (linear growth).

3.4.1 State Representation

Internally, an *edf* represents an evolving data frame with discrete *states*. There are two types of states: *intrinsic states* and *extrinsic states*. Extrinsic states express converging/unbiased estimates; accordingly, they are consumed by downstream *edf*s or other applications, whereas intrinsic states are used to incrementally maintain computed values prior to adjustments and/or estimations.

Suppose we are counting the number of students by their home states. Let *edf*1 represent the dataset we are reading; we have read one out of ten equal-sized partitions, the first partition contains 2 students from IL and 1 student from MI. The intrinsic state α_1 of *edf*1 becomes `[[(id1, IL), (id2, IL), (id3, MI)]]`. For *edf*1, its extrinsic state β_1 is identical to the intrinsic state α_1 because *edf*1 — representing tuples from a base table — requires no adjustments. Let *edf*2 represent `edf1.count(by=state)`; its intrinsic state α_2 become `[[(IL, 2), (MI, 1)]]`. To express unbiased estimates, *edf*2's extrinsic state β_2 is scaled accordingly under the assumption that the unobserved (nine) partitions have the same distribution as the observed (first) partition; thus, β_2 becomes `[(IL, 20), (MI, 10)]`.

We read one more partition (thus, we have read two partitions); the second partition contains 1 student from IL and 1 student from MI. $\alpha_1 = \beta_1$ becomes `[[(id1, IL), (id2, IL), (id3, MI)], [(id4, IL), (id5, MI)]]` (note that the newly added tuples are in a separate list). To (incrementally) update α_2, we first aggregate the second list of β_1, temporarily obtaining `[(IL, 1), (MI, 1)]`, which is merged into α_2 using key-based sum ($\bigoplus$), as described in section 3.2, finally obtaining α_2 `= [(IL, 3), (MI, 2)]`. To obtain unbiased estimates from α_2, we scale individual aggregate values considering the ratio between currently processed tuples and the total tuple count (i.e., 2:10), thereby obtaining *edf*2's extrinsic states β_2 `= [(IL, 15), (MI, 10)]`.

Note that we have taken two different approaches in updating intrinsic states depending on *edf*s. For *edf*1, we have inserted new tuples, creating a longer list for α_1; in contrast, for *edf*2, we have replaced the old set of aggregate values with another set of aggregate values. We systematically distinguish these cases — incremental or complete updates — as follows.

- Intrinsic States: To enable both incremental and complete updates, an *edf*'s states are organized using *versions* and *partials* (a partial is a subset of rows inside each version), as shown in fig. 3.4. Creating a new version means a complete refresh while appending partial(s) to each version (of an *edf*) means incremental updates.

  For example, suppose an *edf*—representing `(first_name, count)` statistics of a class—has a version $\alpha^{(1)}$ and the version currently contains one partial, where the partial has one tuple (e.g., `[(mike, 4)]`). We can incrementally update the version by appending another partial (e.g., `[(sarah, 2)]`); then, the version $\alpha^{(1)}$ represents two tuples `[(mike, 4), (sarah, 2)]`, namely a union of the two partials.

  Specifically, intrinsic states $\boldsymbol{\alpha}$ is a two-dimensional structure (fig. 3.4), consisting of one or more versions $(\alpha^{(1)}, \alpha^{(2)}, \ldots, \alpha^{(v)})$, where each version $\alpha^{(i)}$ contains one or more partials $(\delta_1, \ldots, \delta_p)$. The partials are exclusive from one another with respect to their key; that is, the partials *partition* each version, which is ensured by each *edf* during

operations (section 3.4.2). To obtain the latest intrinsic state, we can union all the partials in the latest version ($\boldsymbol{\alpha}^{(v)}$).

- Extrinsic States: Extrinsic states are introduced to distinguish (external) estimate values from (internal) raw values. In many operations such as map/filter/join, the extrinsic states are simply an alias of intrinsic states since those operations do not need any special adjustments to obtain unbiased estimates. Extrinsic states are required primarily for aggregate operations. There are two types of adjustments.

  The first is when aggregation is *non-mergeable* (e.g., count-distinct), requiring different pre-aggregate representations. Let `edf2 = edf1.count_distinct(name)`, where *edf*1's intrinsic states α_1 consist of two partials δ_1 and δ_1. To incrementally compute count-distinct (i.e., first using δ_1 and then to update it using δ_1), it is insufficient to have the number of unique values appearing in δ_1 because `count_distinct(`δ_1`) + count_distinct(`δ_2`)` is *not* equal to `count_distinct(`$\delta_1 \cup \delta_2$`)`; we need to record all the individual unique values in δ_1 to properly examine if the tuples in δ_2 overlap with any of the values in δ_1. In this case, the intrinsic states must include a set of unique values, which then can be used to incrementally compute count-distinct values (finally appearing in extrinsic states).

  The second type is when aggregate values are expected to increase/change if we observe more tuples in the input: currently observed tuples should be treated as a sample. One example is sum, as we have already described. That is, by treating the current raw summation as the ones from a sample, unbiased estimates can be obtained in consideration of the ratio between the current input cardinality and the projected final input cardinality. This scaling mechanism (called growth-based scaling) is described in section 3.5.

3.4.2 State Transformation

An operation in `edf2 = edf1.op(...)` is a state transformation process from *edf*1's extrinsic states to *edf*2's intrinsic states (which can then be used to produce *edf*2's extrinsic states with optional scaling as described above). In this section, we describe how to transform a version of extrinsic states $\beta_1 = [\delta_1, \ldots, \delta_p]$ *incrementally* to a version of intrinsic states α_2.

- Merge Operation: To incrementally construct α_2 with respect to `op` (when provided $\delta_1, \ldots, \delta_p$ at a time), we exploit the fact that there exists a combination of an intrinsic state representation and a *merge* operation ($\bigoplus$) that can satisfy `op(`$[\delta_1, \ldots, \delta_p]$`)`

edf op	intrinsic repr.	merge ($\bigoplus$)	int. $\rightarrow$ ext.
map	mapped tuples	union	identity
join	joined tuples	union	identity
filter	filtered tuples	union	identity
count	count by key	sum by key	GBI
sum	sum by key	sum by key	GBI
avg	sum/count by key	sum by key	GBI
count_distinct	count by key	sum by key	GBI
min	min by key	min by key	GBI
max	max by key	max by key	GBI
var	sum/count by key	sum by key	GBI
stddev	sum/count by key	sum by key	GBI

Table 3.1: State transformation for each *edf* operation. GBI: growth-based inference.

`= op(`δ_1`)` $\bigoplus$ `...`$\bigoplus$ `op(`δ_p`)`. That is, given δ_1, we can first compute op(δ_1); then, given δ_2, we update the result by merging op(δ_2) into the previous result; this update operation continues for each partial.

For example, suppose we are computing `avg([`δ_1`, `δ_2`, `δ_3`])`, or more specifically, average salary for each state in the United States. To incrementally compute average, we first compute `(count, sum_salary)` for each state from δ_1, which is stored as an intrinsic state. Given the next partial (δ_2), we (again) compute `(count, sum_salary)` for each state from δ_2, then add these aggregates into the earlier results for each state, which is equal to directly computing `(count, sum_salary)` from a union of δ_1 and δ_2. Note that for each `op`, these intrinsic state representations and merge operations differ, which we summarize in table 3.1.

- Primary Key: As described in section 3.3.1, one or more constant attributes serve as a primary key to uniquely identify tuples of an *edf*. Accordingly, our transformation always defines a primary key for a newly created *edf*. `map/filter/join` retains the same key as the input *edf*. Upon `agg`, grouping attributes becomes the key of a new *edf*.

- Clustering Key: A clustering key determines the physical ordering of an *edf*'s tuples. The clustering key changes by aggregation if the aggregation's grouping attributes are not the clustering key itself.

- Base Table Statistics: The *edf* that represents a base table (by reading data from CSV, Apache Parquet, or others) must be provided with (1) a list of file names, (2) the number of tuples in each file, and (3) attributes with primary/clustering keys.

This metadata information is used for computing progress (which affects our inference logic).

- Other Properties: Besides attribute types, a new *edf* must also maintain two internal properties: progress and growth. Since progress is a ratio defined using the original input tuples, every operation simply propagates the progress value to the next *edf* without any modifications. In case of join operations where both the inputs are *edf*s, progress is taken as the minimum of the two progresses. In contrast, growth is newly calculated as part of an operation (each time a new partial or a version is consumed) to accurately estimate the number of tuples that will newly appear in the future.

3.4.3 Properties & Accuracy of Estimates

First of all, all the attribute values produced by DeepOLA are *convergent*. That is, let $\tilde{x}_n$ be an attribute value of an *edf* associated with a certain key after processing up to the n-th tuple, whereas the exact value — the value we obtain after processing the entire data — is x. Then, two properties hold: first, $\mathbb{E}[|\tilde{x}_n - x|] \leq \mathbb{E}[|\tilde{x}_{n'} - x|]$ for $n \leq n'$; and second, $\tilde{x}_N = x$ where N is the total tuple count. While desirable, the latter property ($\tilde{x}_N = x$) is often not ensured by some existing OLA systems that rely on statistical simulations [6].

Moreover, for mean-like aggregates (e.g., count, sum, avg, stddev, var) — possibly before or after map/filter/join — we produce unbiased estimates; that is, $\mathbb{E}[\tilde{x}_n - x] = 0$. In a more complex case such as `avg(c1 * c2)` where both `c1` and `c2` are also results of previous aggregations, the estimates may not be unbiased if `c1` and `c2` are correlated; however, even in this case, the estimates are still convergent. For other aggregates (e.g., count-distinct, extreme order statistics like min/max), we produce reasonably accurate estimates adopting well-known estimation techniques in the literature [38, 39].

3.5 INFERENCE

Given an *edf*'s intrinsic state, we want to generate its extrinsic state. There are three challenges. First, group sizes (e.g., the number of students from a certain state) may grow in a non-linear way as more input data are processed. Second, the number of groups may also increase over time (i.e., the number of states). Third, different types of aggregations often require different estimation mechanisms. To tackle these challenges, our overall inference logic decomposes into two parts: cardinality estimation and aggregate estimation.

3.5.1 Cardinality Estimation

Let $X_i(t)$ denote the ith-group cardinality (i.e. the number of tuples that have been aggregated into the ith-group) at progress t. Various aggregate estimators rely on the current count ($X_i(t)$) and the final count ($X_i(T)$ where $T = 1$) to estimate the final value of an aggregate. Hence, the need to perform cardinality estimation and obtain $X_i(T)$.

DeepOLA models group cardinalities after *monomials* with a shared power, $\mathbb{E}[X_i(t)] \propto t^w$. The underlying reasoning is as followed. DeepOLA assumes that the number of samples and the number of groups follow two hidden monomials, $\mathbb{E}[n(t)] \propto t^u$ and $\mathbb{E}[m(t)] \propto t^v$, respectively. Then, average group cardinality is $\frac{1}{m(t)} \sum_{i=1}^{m(t)} X_i(t) = \frac{1}{m(t)} n(t)$ whose expectation is proportional to t^{u-v}, so $w = u - v$. A naive way to perform cardinality estimation would be to directly divide by the progress. This formulation also captures this trivial solution with $w = 1$. This modeling along with these captures many other scenarios in Deep OLA. For example, if the input data frame is a table reader, then DeepOLA would expect the sample to grow linearly ($u = 1$). If the input is behind a cross join of two tables, then DeepOLA would expect a quadratic growth ($u = 2$). Filtering would then affect the coefficient corresponding to its selectivity. On the other hand, if the group key is the same as the clustering key, DeepOLA would see the number of groups grows linearly ($v = 1$) as it consumes more partitions. A low-cardinality group key would result in a constant ($v = 0$) while a higher-cardinality one would generate something in between ($0 < v \leq 1$). Furthermore, this model simplifies its estimation logic. In fact, DeepOLA does not need to estimate $\mathbb{E}[n(t)]$ nor $\mathbb{E}[m(t)]$, but only $\mathbb{E}[X_i(t)]$.

DeepOLA fits the power w in a logarithmic-transformed linear regression. The regression is fit in a streaming fashion with O(1) time/space complexities per observation.

3.5.2 Aggregate Estimation

Depending on the aggregate function type, DeepOLA selects an aggregate estimator from the following. Note, $y_{i,:t}$ denotes the aggregate value of the ith-group at progress t, $x_{i,:t}$ denotes the observed cardinality of the ith-group at progress t and $\hat{x}_{i,:t}$ denotes the estimate of final cardinality at progress t.

- Count: Use the estimated cardinality.

$$f_{\text{count}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = \hat{x}_{i,:t} \tag{3.1}$$

- Sum: Scale the summation.

$$f_{\text{sum}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = \frac{y_{i,t}}{x_{i,t}} \hat{x}_{i,:t} \tag{3.2}$$

- Weighted Average: Weighted averages (e.g., average, variance, standard deviation) are special cases of summation. Because of our choice of estimators, average estimators reduce to the identity function. Let $y'_{i,:t}$ be the weighted summation, $y''_{i,:t}$ be the summation of weights and $y_{i,t} = y'_{i,t}/y''_{i,t}$ be the weighted average:

$$f_{\text{avg}}((y'_{i,:t}, y''_{i,:t}), x_{i,:t}, \hat{x}_{i,:t}) = \left(\frac{y'_{i,t}}{x_{i,t}} \hat{x}_{i,:t}\right) / \left(\frac{y''_{i,t}}{x_{i,t}} \hat{x}_{i,:t}\right) = y_{i,t} \tag{3.3}$$

- Count Distinct: DeepOLA adopts a finite-population method-of-moment estimator [38] (denoted as $\hat{D}_{MM1}$). For brevity in this subsection, let us focus on i-th group and shorten the notations of current group cardinality $x = x_{i,t}$, final estimated group cardinality $X = \hat{x}_{i,:t}$, and current group count distinct $y = y_{i,t}$. DeepOLA computes $f_{cd}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = Y$ where Y satisfies eq. (3.4).

$$y_{i,t} = Y(1 - h(\hat{x}_{i,:t}/Y)) \tag{3.4}$$

  $h(z)$ is defined below. To solve the equation, DeepOLA runs Newton-Raphson iterations until convergence with a tolerance and at most a finite number of steps. Each iteration involves evaluating numerical approximation of gamma and digamma functions.

$$h(z) = \frac{\Gamma(\hat{x}_{i,:t} - z + 1)\,\Gamma(\hat{x}_{i,:t} - x_{i,t} + 1)}{\Gamma(\hat{x}_{i,:t} - x_{i,t} - z + 1)\,\Gamma(\hat{x}_{i,:t} + 1)} \tag{3.5}$$

- Order Statistics: Order statistics include min, max, median, quantiles, and k-th smallest/largest values. Currently, DeepOLA simply outputs the latest value:

$$f_{\text{order}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = y_{i,t} \tag{3.6}$$

  which provides a fairly accurate estimate for large $\hat{x}_{i,:t}$ at no computation cost.

CHAPTER 4: IMPLEMENTATION

This chapter describes the implementation details of DeepOLA, a Rust-based implementation of the novel online-aggregation data model described earlier. The model is implemented in Rust owing to the high performance, type-safety and memory-safety properties of the Rust language.

The implementation module can be divided into two parts, (1) Query Service - that lets the user build a query, and (2) Execution Engine - that lets the user execute the query on a given input data and obtain incremental online results.

4.1 QUERY SERVICE

The Query Service lets the user represent a query that they want to evaluate as an execution graph, composed of different *nodes* — representing the various operations, and *edges* — representing the data flow paths between these nodes. A node can have multiple incoming edges depending on the type of the operation. For example, consider the sequence of example *edf* operations shown in Figure 4.1a. Figure 4.1b shows the execution graph with the corresponding nodes.

Each node implements a `process()` method that takes the input message i.e. an *edf* (from one or more input channel(s)), updates the node's internal state and generates the output for the downstream nodes to be able to consume. The *edges* are implemented using `channels` — a message-passing based mechanism for sending a stream of messages across threads. Each edge defined in the query service graph, gets a corresponding channel, where the source node (of the edge) writes to the channel and the destination node (of the edge) reads from the channel.

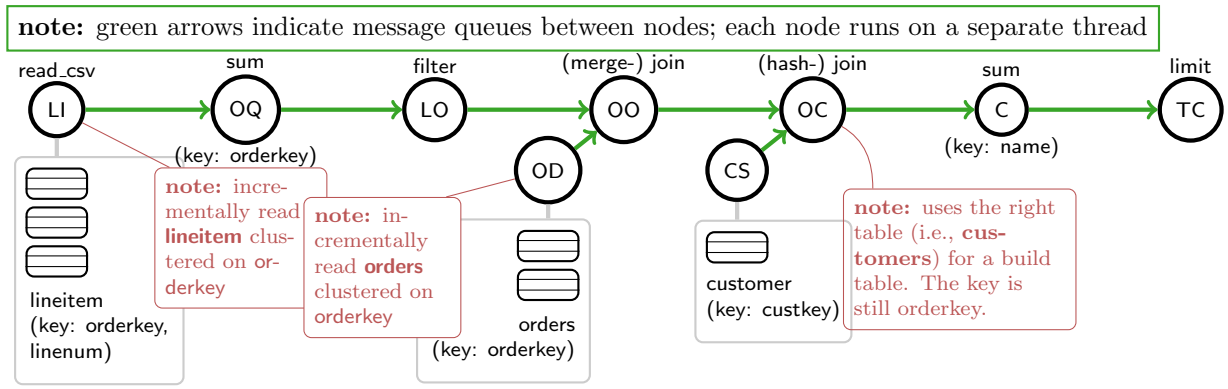
Node	Num of inputs	Operations
Reader	1	To read partitioned input data from underlying storage
Appender	1	To perform `map` and `filter` operations
Accumulator	1	To perform accumulation operations like `GROUP-BY`
Join	2	To perform sort/merge join on two tables
Merger	2	To perform custom merge operations on two tables

Table 4.1: Different types of nodes supported in DeepOLA

Table 4.1 shows the set of nodes implemented in DeepOLA along with the number of inputs and the actual operation performed by the node. These nodes combined with differ-

```
lineitem = read_csv('...')          # LI
# item count for each order
order_qty = lineitem.sum(qty, by=orderkey)     # OQ
# select only the large orders
lg_orders = order_qty.filter(sum_qty > 300)    # LO
# find the customers with biggest order sizes
lg_order_cust = lg_orders.join(orders) \       # OO
                  .join(customer)              # OC
# select top-100 customers
top_cust = lg_order_cust.sum(sum_qty, by=name) \ # C
             .limit(100)  # TC
```

(a) An example sequence of data operations with *edf*



(b) DeepOLA's internal representation to process above example in parallel

Figure 4.1: An example sequence of operations and its corresponding execution graph.

ent partition-processing mechanism allows DeepOLA to express a large set of SQL queries (including all the 22 TPC-H benchmark queries).

The user can create instances of different types of nodes and then connect them through different edges. These nodes and edges put together constitute a query service. An output reader node is connected to the final operation node to generate the query output.

4.2 EXECUTION ENGINE

The Execution Engine takes a Query Service and evaluates the query on a specified dataset generating a sequence of *edf* outputs. The `reader` nodes serve as the root nodes of the execution graph — defining the dataset on which the query is to be evaluated. Once the inputs for the `reader` nodes are defined, the execution engine can start the query execution.

Once the query execution is started, each node in the Query Service runs in a separate thread. Each node, waits till it receives a message on its input channel(s). Once a message

is read from the input channels, it starts processing the message. A message consists of the following: (1) a shared pointer to a dataframe, and (2) a metadata containing information on the progress of the query execution. For the `reader` nodes, the initial dataframe is a list of filenames that needs to be read as separate partitions. Currently, DeepOLA supports `CSV` and `Apache Parquet` format for reading files. This can be extended to support other file formats.

The node processes these messages, updates its intrinsic states using the metadata and the processing logic explained in section 3.4, generates a dataframe corresponding to its extrinsic state and writes it along with the updated metadata as a message to the output channels. In case there are no messages on a node's input channel, the node blocks on the channel read. A special message type — EOF — is used to indicate the end of inputs on a given channel. Once an execution node receives EOF on its input channels, the node sends an EOF message on the output channels indicating termination of the node's execution. Note, nodes with multiple input channels can have different behavior depending upon whether the execution needs to terminate on EOF on one or all channels. The user can define such a behavior depending on the specific operation's requirement.

4.3 DISCUSSION

The online query processing model allows DeepOLA to take advantage of various optimizations that help in significant query processing speedup for online processing. Some of these optimizations are:

- Pipelined Query Operations: Each node's operation runs in a separate thread and can be processing different partitions of data at different time instance. Hence, we use a pipelined implementation where a node once finishes processing a partition can start processing the next partition irrespective of whether the earlier partition has been fully consumed or not by the remaining nodes.

- Online Sort-Merge Join: When both the input tables are sorted on a clustering key, which is a superset of the join keys, the tables can be merged online with negligible overhead. For example: consider `lineitem` and `orders` table from TPC-H schema, clustered on `l_orderkey`. A join on this column can be evaluated in an online manner without additional overhead.

- Reusing Hash Table: When evaluating a join operation where we are building a hash table for the right table, to avoid re-computation for each partial join, the hash ta-

ble can be materialized and stored in-memory, and then re-used for all partial join computations.

- Shared Data Pointers: DeepOLA instead of duplicating dataframe, sends shared pointers of dataframe across different nodes (and threads) thereby reducing the cloning costs significantly.

Various potential optimizations like pruning the execution graph of nodes that aren't consumed, pushing down filter nodes closer to the reader nodes, combining multiple map/filter operations in a single node instead of running them serially, etc. can further optimize DeepOLA's performance. Currently, DeepOLA doesn't contain a mechanism to automatically convert a SQL-style query to a query service. A user thus has to build the query execution graph and consider query plan related optimizations such as order of join operations, pruning unnecessary columns, pushing down predicates, etc. Building a system that offers these optimizations with a SQL-like declarative interface is a natural extension of DeepOLA.

CHAPTER 5: EVALUATION

In this chapter, we describe the various experiments performed to evaluate the generalizability and performance of DeepOLA compared to various existing exact-query and online-aggregation baselines. Our experiments on a 100 GB TPC-H benchmark dataset show the following findings:

- DeepOLA's first estimate is 12.8× faster than the exact systems while being 1.3× slower in producing exact results. (section 5.2)
- DeepOLA's first estimates have a median error of 2.70%. DeepOLA provides results with under 1% error in 3.17× fraction (upto 48.80× for some queries) of the best baseline exact query processing systems' latency. (section 5.3)
- DeepOLA produces the results with less than 1% error, 1.92× times faster than state-of-the-art OLA systems. (section 5.4)
- DeepOLA's performance can be further improved by optimally choosing the partition sizes. (section 5.5.1)
- DeepOLA's pipelined execution of *edf* operations giving significant speed over other exact query processing systems. (section 5.5.2)

5.1 EXPERIMENTAL SETUP

All the experiments discussed in this chapter are performed on a Standard D16ads v5 (Azure) machine with 16 vCPU(s) and 64 GB of memory. The TPC-H benchmark is used to compare and evaluate the performance. A scale `s=100` (i.e. 100 GB) TPC-H dataset is generated using the TPC-H data generation kit. For DeepOLA, the dataset is then partitioned into different partitions of size 512 MB each, which are then converted into Apache Parquet format.

5.1.1 Baselines

The performance of DeepOLA is compared against the following online aggregation and exact query processing baselines:

1. ProgressiveDB [5] - A middleware-based OLA system that provides a SQL database the capability to perform online aggregation. The implementation [40] provided by

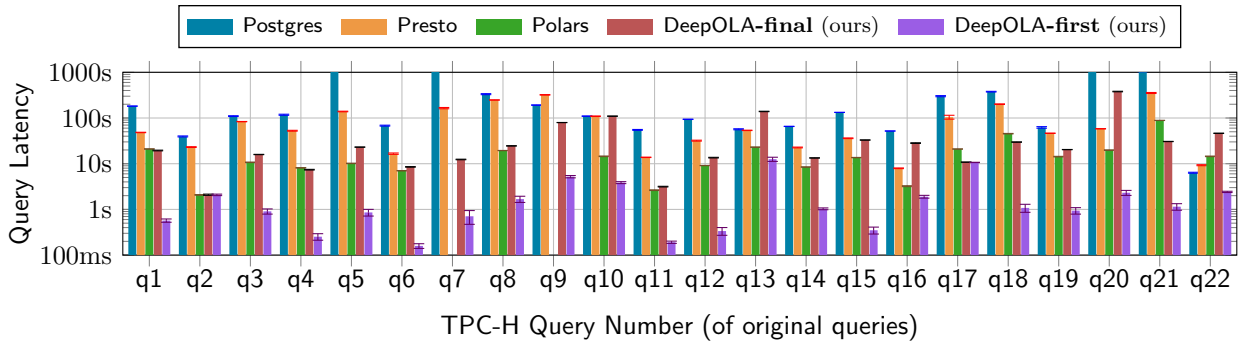


Figure 5.1: Comparison of different baselines on TPC-H 100 GB dataset.

the authors, currently limited to single-table queries (non-nested, join-free), is used to compare the performance on queries Q1 and Q6 from the TPC-H benchmark.

2. WanderJoin [6] - A random-walk based OLA system for nested, multi-join queries. The implementation [41] provided by the authors, along with the modified TPC-H queries (Q3, Q7, Q10) is used for performance comparison.

3. Presto [7] - A data warehouse designed for diverse data sources. It provides a SQL interface to query big data. We set up Presto in a single-node mode, using the hive connector on HDFS.

4. Postgres - A popular RDBMS. We create appropriate foreign-keys and indexes on the attributes according to the TPC-H schema.

5. Polars [8] - A dataframe library in Rust, providing a Pandas-like dataframe interface. Polars adopts various optimizations like SIMD, Apache Arrow, lock-free parallel hashing, etc.

5.2 QUERY PROCESSING OVERHEAD

In this experiment, DeepOLA's latency and memory usage is compared to different exact query processing systems - Presto, Postgres and Polars. All the 22 TPC-H queries are run on the 100 GB dataset, and their overall query processing time is noted. For Polars, we also measure its memory usage, as it is an in-memory dataframe processing library.

Figure 5.1 shows the time taken by DeepOLA to obtain the first and the final result. The number of intermediate results depends on the number of partitions of the tables used in the query. DeepOLA obtains first estimates 11.8× faster than Polars' exact answers, 78.3× faster than Presto's exact answers and 238.3× faster than Postgres' exact answers.

In terms of the slowdown for the final result, measured as the ratio of DeepOLA's final result latency and other baseline's final result latency, the median slowdown against Polars is 1.5×, against Presto is 0.3× and against Postgres is 0.1×, meaning DeepOLA, on median, produces exact answers faster than Presto and Postgres. Despite being an online aggregation system, DeepOLA produced final result faster than Postgres in 22 of the queries, than Presto in 17 queries and than Polars in 6 queries.

Moreover, DeepOLA has low peak memory utilization. Polars runs out of memory (on the experiment machine) for queries Q7 and Q9, not able to generate any results, whereas DeepOLA successfully produced initial estimates and the exact query results. On average, DeepOLA's peak memory usage is 4.3× less than Polars (up to 17.4× less for some queries), providing the ability to handle larger datasets for exact query processing.

Specifically looking at some of the queries, Q9, Q10, and Q13 require building hash tables for smaller right tables before being able to produce first-result, thus have smaller improvement. Q2 and Q17 require computing sub-queries' aggregate and thus have negligible gains (but almost zero overhead). In terms of total query latency, computational overhead is most prominent in Q10 and Q13 (due to repeated group-by on high-cardinality `c_custkey`) and Q20 (due to repeated filter on `partsupp`).

5.3 QUERY APPROXIMATION ERROR

In this experiment, we analyze approximation errors in DeepOLA's OLA outputs (as it processes more data over time) in terms of MAPE and recall error. Figure 5.2 shows time-error curves for a few representative queries in three different categories, as follows.

The first category includes queries on non-clustering group-by keys with low cardinality. Overall, their MAPE curves decrease over time as DeepOLA observes more data while recalls reach 100% early on. Many queries in TPC-H falls into this category: Q1, Q4–Q9, Q12, Q14, Q17, Q19, and Q22. In particular, Q8 (Figure 5.2-top-left) involves a weighted average group-by aggregation over multiple joined tables. DeepOLA is able to answer the first estimate at 1.9s with 6.5% error. When Polars completes (at 19s), DeepOLA has 0.87% error. The query Q9 also (Figure 5.2-bottom-left), show a similar behavior. Polars fails to evaluate the query because of memory overflow, whereas DeepOLA obtains the first result at 6.09s with 2.03% error.

The queries in the second category involves clustering group-by keys; therefore, their aggregation values are exact (MAPE at 0%) while their recall increases as DeepOLA retrieve keys from different partitions. Q3, Q18 (Figure 5.2-top-right), and Q20 are examples: their recalls increase linearly as more keys are retrieved/observed.

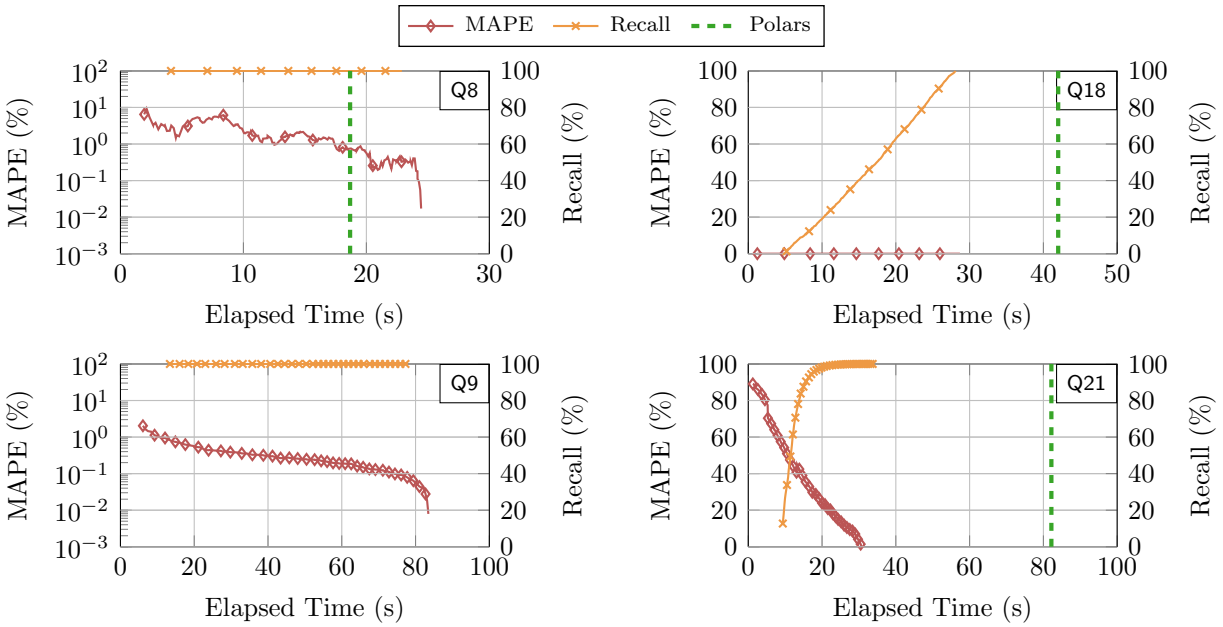
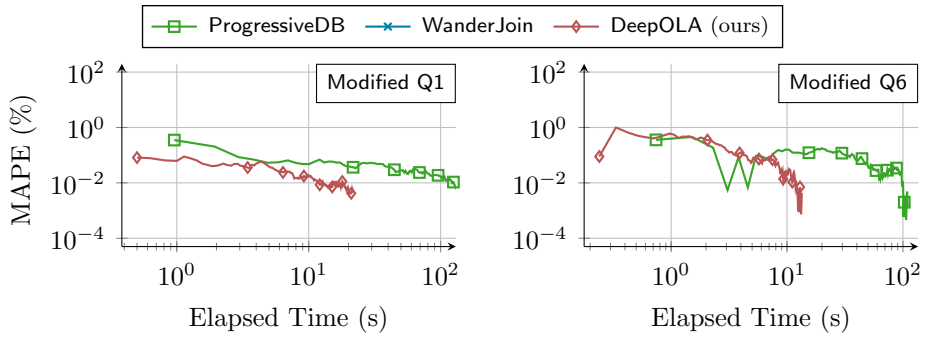


Figure 5.2: DeepOLA's approximation error measured in mean absolute percentage error (MAPE) and recall over time for some representative queries. Vertical lines represent completion time of exact methods. From top to bottom, left to right (Q8, Q9, Q18, Q21), Postgres completes in (332s, 191s, 376s, 1061s), while Presto completes in (247s, 321s, 200s, 352s) respectively. Note: Polars isn't able to completely evaluate Q9 due to memory overflow.

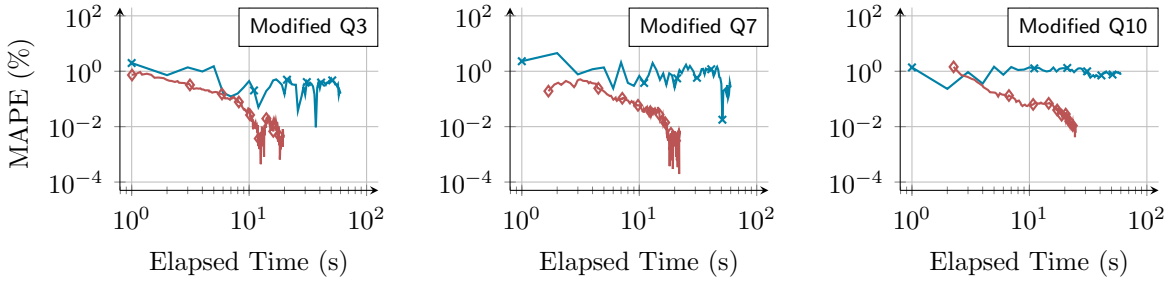
The third category is a combination of the first two; that is, their errors can be understood with MAPE, recall, and/or precision. For instance, Q10, Q16, and Q21 (Figure 5.2-bottom-right) have quickly rising recall curves while their MAPE curves drop only linearly because their group-by keys are diverse, leading to lower number of samples per group and so lower prediction power. Q11 has a perfect MAPE score but its recall/precision curves increase quickly toward the end. Q2 and Q15 have on-off recall and precision due to their uses of arguments of the minima and maxima. Finally, Q13 aggregates counts over a high-cardinality non-clustering key (`c_custkey`), followed by an outer aggregation over the inner count as the group-by key. Because the inner count changes over different partitions, the growth within outer groups can be non-monotonic, violating DeepOLA's cardinality estimator. Consequently, its MAPE is relatively large compared to those of other queries.

5.4 COMPARISON WITH EXISTING OLA

In this experiment, we compare DeepOLA against other OLA systems: ProgressiveDB and WanderJoin. Since ProgressiveDB and WanderJoin supports only a limited set of TPC-H



(a) vs ProgressiveDB (using single-table Q1 and Q6)



(b) vs WanderJoin (using same modified Q3, Q7, and Q10 as in [6])

Figure 5.3: Comparison of approximation error over time against other OLA systems

queries, our evaluation against them use those queries; specifically, Q1 and Q6 for ProgressiveDB, and Q3, Q7, Q10 for WanderJoin.

Figure 5.3 shows the results from ProgressiveDB on Q1 and Q6. Although the initial estimates of ProgressiveDB and DeepOLA are close, DeepOLA converges 2.5× faster than ProgressiveDB to a less than 1% relative error. Figure 5.3b shows the comparison against WanderJoin. Although the errors of the first estimate are comparable, the convergence of DeepOLA to a less than 1% relative error is 1.51× faster than WanderJoin. Moreover, DeepOLA soon converges to exact answers whereas WanderJoin stays around 1% relative errors, which are expected because its random walk-based sampling mechanism is not designed in such a way. We believe the approach taken by DeepOLA — converging to exact answers — is more desirable for end users.

5.5 FURTHER ANALYSIS

In this section, we look at a few other experiments performed to try and understand DeepOLA's performance.

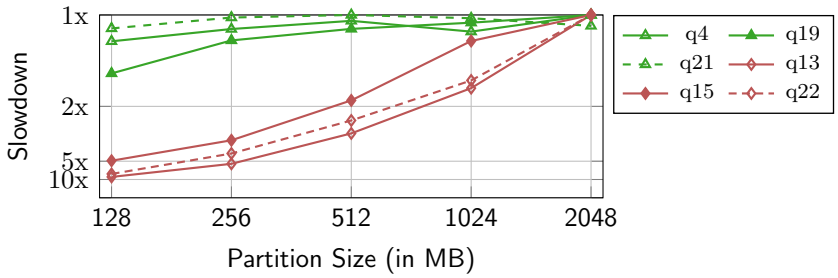


Figure 5.4: The impact of partition size on DeepOLA's performance. The queries shown in red — q13, q15, q22 — incur higher merge costs than the ones in green — q4, q19, q21.

5.5.1 Data Partition Size

To understand the impact of partition sizes on overall query latencies, we evaluate DeepOLA on scale-100 (i.e. 100 GB) data with different partition sizes (128 MB, 256 MB, ..., 2048 MB). As individual partition sizes increase, the time taken to generate the first-result increases whereas the final-result latency tends to decrease because the overhead of merging multiple partitions is lower. Figure 5.4 shows the latencies of multiple TPC-H queries as a multiple of the best performance observed for that query across different partitions.

For queries where the merge operation overhead is small, the impact of partition size is negligible, and we observe similar final-result latency across partition sizes. Some example queries are Q1, Q4, Q6, Q7, Q12 and Q19 (group-by-agg has few groups) and Q18 and Q21 (streamed on o_orderkey).

For queries involving operations with higher merge overhead like group-by with large number of groups, there is a significant difference in performance across partition sizes. A larger partition size performs better on final-result latency as the number of partitions decrease and thus the computation overhead of re-merging groups drops. Some example queries are Q13, Q15 and Q16 (group-by-agg has large number of groups), and Q22 (pruning of c_custkey).

For less OLA-friendly queries (e.g., Q17), having a larger partition size helps in reducing the final-result latency, and thus also the first-result latency. Hence, depending on the query, a suitable partition size might be chosen to optimize the analysis goal — be it either first-result latency or final-result latency. This work, however, does not investigate such an optimizer.

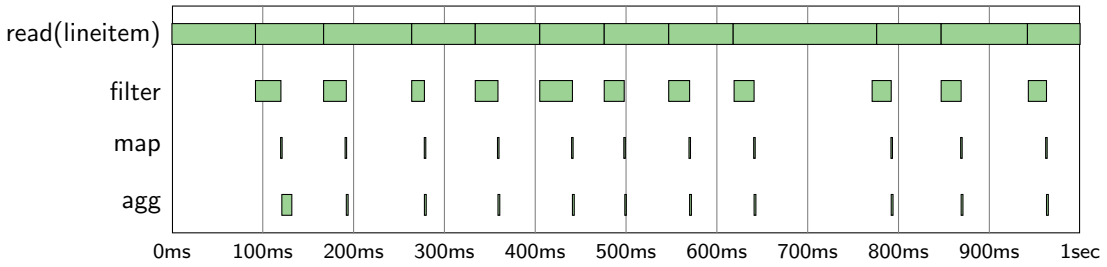


Figure 5.5: Pipelined execution of example query Q6

5.5.2 Pipelined Query Processing

A key factor for DeepOLA's total query latency to be comparable or lower than other exact processing systems despite additional overhead of merge computations is pipelining of different operations. Since each operation, given its input *edf*, can operate independently, a pipelined execution leads to higher utilization of available compute, reducing the total time taken.

In Figure 5.5, we plot how different nodes process data partitions over the execution of a query. Although the query executed for a larger duration, we limit the observed timeline to 1 second for the sake of clarity. The operation `agg` is the final node for query Q6. Thus, each block in the timeline of operation `agg` represents an obtained output.

Such a view on the internal query processing also helps in identifying the bottleneck operation, thereby providing the user with feedback to see what operations they should try to optimize.

CHAPTER 6: CONCLUSION

In this work, we take a step toward Deep OLA (Online Aggregation), by introducing a novel data model that is *closed* under set-oriented operations (e.g., map/filter/join/agg), thus enabling applications of nested operations to previous OLA outputs.

We show its viability through DeepOLA— a Deep OLA system implemented in Rust. We evaluated DeepOLA on TPC-H (100 GB) by comparing against state-of-the-art OLA engines as well as conventional data systems. Our experiments show that DeepOLA provides first estimates 12.8× faster (median) than conventional systems computing exact answers while having (only) a median 2.70% relative error. Moreover, DeepOLA incurs a very small overhead (1.3× median slowdown) in producing exact answers. In fact, the pipelined implementation of different ops in DeepOLA often provides faster total latencies than exact query processing engines for some queries. In the future, we aim to extend DeepOLA to support a SQL-like declarative interface with automated query optimizations. We also aim to extend DeepOLA to a distributed setup, making deep online aggregation further scalable.

REFERENCES

[1] N. Sheoran, "DeepOLA: Online Aggregation for Deeply Nested Queries," in *Proceedings of the 2022 International Conference on Management of Data*, 2022, pp. 2527–2529.

[2] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE transactions on visualization and computer graphics*, vol. 20, no. 12, pp. 2122–2131, 2014.

[3] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," in *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, 1997, pp. 171–182.

[4] W. McKinney et al., "Pandas: a foundational Python library for data analysis and statistics," *Python for high performance and scientific computing*, vol. 14, no. 9, pp. 1–9, 2011.

[5] L. Berg, T. Ziegler, C. Binnig, and U. Röhm, "ProgressiveDB: progressive data analytics as a middleware," *Proceedings of the 45th International Conference on Very Large Data Bases*, vol. 12, no. 12, pp. 1814–1817, 2019.

[6] F. Li, B. Wu, K. Yi, and Z. Zhao, "Wander join: Online aggregation via random walks," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 615–629.

[7] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: SQL on Everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1802–1813.

[8] pola-rs, "Polars: Lightning-fast DataFrame library for Rust and Python," https://www.pola.rs/, Accessed: 2022-11-27.

[9] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, 1999, pp. 287–298.

[10] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton, "A scalable hash ripple join algorithm," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, 2002, pp. 252–262.

[11] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica, "G-ola: Generalized online aggregation for interactive analysis on big data," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015, pp. 913–918.

[12] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding, "Quickr: Lazily approximating complex adhoc queries in bigdata clusters," in *Proceedings of the 2016 international conference on management of data*, 2016, pp. 631–646.

[13] Y. Park, B. Mozafari, J. Sorenson, and J. Wang, "VerdictDB: Universalizing approximate query processing," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 1461–1476.

[14] S. Chaudhuri, B. Ding, and S. Kandula, "Approximate Query Processing: No Silver Bullet," in *Proceedings of the 2017 ACM International Conference on Management of Data*, ser. SIGMOD '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3035918.3056097 p. 511–519.

[15] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine et al., "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches," *Foundations and Trends in Databases*, vol. 4, no. 1–3, pp. 1–294, 2011.

[16] S. Chaudhuri, G. Das, and V. Narasayya, "Optimized stratified sampling for approximate query processing," *ACM Transactions on Database Systems (TODS)*, vol. 32, no. 2, pp. 9–es, 2007.

[17] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic Sample Selection for Approximate Query Processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003, pp. 539–550.

[18] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica, "BlinkDB: queries with bounded errors and bounded response times on very large data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, 2013, pp. 29–42.

[19] K. Chakrabarti, M. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," *Proceedings of the 27th International Conference on Very Large Databases*, vol. 10, no. 2, pp. 199–223, 2001.

[20] Y. E. Ioannidis and V. Poosala, "Histogram-based approximation of set-valued query-answers," *Proceedings of the 25th International Conference on Very Large Databases*, vol. 99, pp. 174–185, 1999.

[21] V. Poosala, V. Ganti, and Y. E. Ioannidis, "Approximate query answering using histograms," *IEEE Data Eng. Bull.*, vol. 22, no. 4, pp. 5–14, 1999.

[22] G. Cormode, "Sketch techniques for approximate query processing," *Foundations and Trends in Databases. NOW publishers*, p. 15, 2011.

[23] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: Learn from Data, not from Queries!" *Proceedings of the 45th International Conference on Very Large Databases*, vol. 13, no. 7, 2019.

[24] Q. Ma and P. Triantafillou, "DBEst: Revisiting Approximate Query Processing Engines with Machine Learning Models," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1553–1570.

[25] N. Park, M. Mohammadi, K. Gorde, S. Jajodia, H. Park, and Y. Kim, "Data synthesis based on generative adversarial networks," *arXiv preprint arXiv:1806.03384*, 2018.

[26] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni, "Modeling Tabular data using Conditional GAN," in *Advances in Neural Information Processing Systems*, 2019.

[27] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das, "Approximate query processing for data exploration using deep generative models," in *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 2020, pp. 1309–1320.

[28] M. Zhang and H. Wang, "Approximate query processing for group-by queries based on conditional generative models," *arXiv preprint arXiv:2101.02914*, 2021.

[29] N. Sheoran, S. Mitra, V. Porwal, S. Ghetia, J. Varshney, T. Mai, A. Rao, and V. Maddukuri, "Conditional Generative Model Based Predicate-Aware Query Approximation," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 8, pp. 8259–8266, Jun. 2022.

[30] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "Progressive merge join: a generic and non-blocking sort-based join algorithm," in *Proceedings of the 28th international conference on Very Large Data Bases*, 2002, pp. 299–310.

[31] C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol, "The sort-merge-shrink join," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 4, pp. 1382–1416, 2006.

[32] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra, "Scalable approximate query processing with the dbo engine," *ACM Transactions on Database Systems (TODS)*, vol. 33, no. 4, pp. 1–54, 2008.

[33] "TPC-H: Decision Support Benchmark," https://www.tpc.org/tpch/, Accessed: 2022-11-27.

[34] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 2007, pp. 137–156.

[35] "MySQL 8.0 Reference - GROUP_CONCAT," https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_group-concat, Accessed: 2022-11-27.

[36] "MySQL 8.0 Reference - FIND_IN_SET," https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_find-in-set, Accessed: 2022-11-27.

[37] I. Hellström, "Oracle SQL & PL/SQL Optimization for Developers," https://oracle.readthedocs.io/en/latest/sql/joins/hash-join.html, Accessed: 2022-11-27.

[38] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, "Sampling-Based Estimation of the Number of Distinct Values of an Attribute," *Proceedings of the 21st International Conference on Very Large Databases*, pp. 311–322, 1995.

[39] A. W. v. d. Vaart, *Asymptotic Statistics*, ser. Cambridge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, 1998.

[40] "ProgressiveDB," https://github.com/DataManagementLab/progressiveDB, Accessed: 2022-11-27.

[41] "XDB: approXimate DataBase (XDB)," https://github.com/InitialDLab/XDB, Accessed: 2022-11-27.