# A Step Toward Deep Online Aggregation

NIKHIL SHEORAN*†, Databricks, USA
SUPAWIT CHOCKCHOWWAT†, University of Illinois at Urbana-Champaign, USA
ARAV CHHEDA, University of Illinois at Urbana-Champaign, USA
SUWEN WANG, University of Illinois at Urbana-Champaign, USA
RIYA VERMA, University of Illinois at Urbana-Champaign, USA
YONGJOO PARK, University of Illinois at Urbana-Champaign, USA

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 Wake, an edf-based OLA system, by examining against state-of-the-art OLA and non-OLA systems. In our experiments on TPC-H dataset, Wake produces its first estimates 4.93× faster (median)—with 1.3× median slowdown for exact answers—compared to conventional systems. Besides its generality, Wake is also 1.92× faster (median) than existing OLA systems in producing estimates of under 1% relative errors.



CCS Concepts: • **Information systems → Database query processing**; **Online analytical processing engines**; *Relational parallel and distributed DBMSs*; **Uncertainty**; **Relational database model**; • **Mathematics of computing →** *Time series analysis*; • **Theory of computation → Streaming models**.

Additional Key Words and Phrases: online aggregation, nested query, SQL, data frame, evolving data frame, time-series forecasting, cardinality estimation, confidence interval

**ACM Reference Format:**
Nikhil Sheoran, Supawit Chockchowwat, Arav Chheda, Suwen Wang, Riya Verma, and Yongjoo Park. 2023. A Step Toward Deep Online Aggregation. *Proc. ACM Manag. Data* 1, 1, Article 124 (June 2023), 28 pages. https://doi.org/10.1145/3589269

*Work done while at the University of Illinois at Urbana-Champaign.
†These authors contributed equally to this work.

Authors' addresses: Nikhil Sheoran, nikhil.sheoran@databricks.com, Databricks, USA; Supawit Chockchowwat, supawit2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Arav Chheda, aravmc2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Suwen Wang, suwenw2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Riya Verma, rverm2@illinois.edu, University of Illinois at Urbana-Champaign, USA; Yongjoo Park, yongjoo@illinois.edu, University of Illinois at Urbana-Champaign, USA.


2836-6573/2023/6-ART124 $15.00
https://doi.org/10.1145/3589269

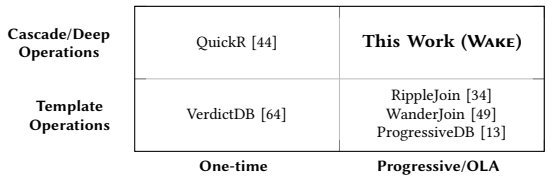



Fig. 1. Our system Wake enables OLA for deep operations

## 1 INTRODUCTION

For ad-hoc data exploration, the fastest way to gain insights would be to extract as much useful information as possible from partial data processing by computing intermediate estimates that highly resemble the (future) final answer, while continuously refining the estimates until the entire data is processed. Since the pioneering work by Hellerstein et al. [38], this data processing paradigm—called *Online Aggregation* (OLA)—has been studied in various directions to improve its generality and performance with new architectures [13, 60, 84], novel join algorithms [21, 29, 34, 49, 52, 57, 81], support of subqueries [90], specialized indexing [27, 45, 63, 85], etc.

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, suppose a data analysis session[1] expressed in pandas-like methods [54]:

```
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 qty_per_cust = lg_order_cust.sum(sum_qty, by=name)
9 top_cust = qty_per_cust.sort(sum_qty, desc=True).limit(100)
```

The first output (L3) is aggregated/filtered again to find the top customers (L5-L9). 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 new data/processing model designed to enable Deep Online Aggregation—the ability to apply subsequent operations to previous OLA outputs for another OLA output.* For each operation, edf offers *converging* estimates for the final answer—with diminishing expected errors (§4.5)—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 systems. To evaluate the viability of our approach, we implement Wake[2], an edf-based OLA system, and examine its performance against existing OLA systems (ProgressiveDB [13], WanderJoin [49]) as well as modern data systems (Presto [74],

[+]Work done while at University of Illinois at Urbana-Champaign.
[*]These authors contributed equally to this work.
[1]This example data analysis is a rewritten version of TPC-H query 18.
[2]Wake stands for **W**e **A**lready **K**now **E**nough.

Table 1. Summary of existing work. ▲/✗ indicates limited/no support.

| System/Method | OLA? | DeepQ? | Novelty | Weakness/Difference |
|---|---|---|---|---|
| OLA [38] | ✔ | ✗ | The first OLA proposal | Only for simple SQL with no joins/subqueries |
| RippleJoin [34, 52] | ✔ | ▲ | Join algorithm for OLA | Exponential complexity for multiple joins |
| WanderJoin [49] | ✔ | ▲ | Supports multiple joins | Requires indexes / May not produce exact answers |
| G-OLA [90] | ✔ | ▲ | Supports filters with subqueries | Some data need repetitive processing |
| QuickR [44] | ✗ | ✔ | Pushes down sampling operators | Not OLA (each query answer is from a single sample) |
| VerdictDB [64] | ✗ | ▲ | Platform-independent | Not OLA (each query answer is from a single sample) |
| **Ours (Wake)** | ✔ | ✔ | Supports deeply nested operations | May need more memory (§4); need to tune partition size (§8.7); no query optimizer |

PostgreSQL, Polars [69]). *To our knowledge, no previous work has formally studied a data model for Deep OLA and has evaluated its efficiency for practical use cases with comprehensive experiments.*

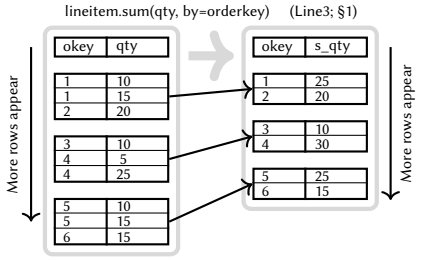
***Challenges.*** Given a query (or an operation), existing OLA can be understood as a process that converts an input data into a series of intermediate/final results, where notably, the input and the output are of different types, which is the fundamental reason that OLA cannot be applied to the results of OLA. Specifically, we observe the following challenges. First, the existing model for structured data (which we call *data frame*) 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.

***Our 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., *mutable attributes* and *cardinality growth*, which are key to producing accurate estimates (§2.3). Third, our internal processing is designed to minimize redundant computation whenever possible.

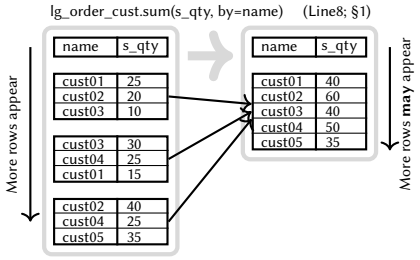
***Orthogonal Work.*** OLA (including Deep OLA) can be understood as a mechanism that translates aggregation-involving queries into an incremental computation logic, making it orthogonal to incremental computation frameworks [55, 58] and incremental view maintenance [10, 91]. OLA belongs to Approximate Query Processing, a broader class of query processing paradigm; for example, sampling-based AQP [8, 44, 64] produces a single approximate answer (not a series of continuously refining answers like OLA).

***Contributions.*** This paper shares the following findings:

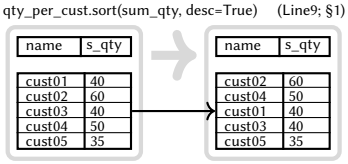
(1) Our new data model, *evolving data frames* (edf), enables successive OLA operations. (§3)
(2) Our processing model, representing common set operations, can transform an edf into another edf (with correct properties) relying on our internal inference technique. (§4 and §5)
(3) Our extended data model allows propagating confidence intervals through the pipeline. (§6)
(4) Wake's multi-thread implementation for OLA can offer high processing speed with pipelined parallelism. (§7)




(a) Case 1: Order-preserving Local Operation




(b) Case 2: Shuffle Op with Inference




(c) Case 3: Shuffle Op w/o Inference

Fig. 2. 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.

(5) Wake can produce an intermediate result 4.93× faster than the associated final answer, while Wake incurs (only) 1.3× overhead compared to non-OLA. The median relative accuracy of the first answer for TPC-H queries is 2.70%, which converges quickly toward zero. (§8)

The technical contributions listed above appears after we present our motivation behind the design of evolving data frames (§2).

## 2 MOTIVATION

We first describe a need for Deep OLA-specific data model (§2.1). To achieve Deep OLA, we present several cases our proposed framework must handle (§2.2), and discuss new unique properties (§2.3).

### 2.1 OLA Input/Output As Type

We argue that it is critical to formalize the outputs of OLA as a *type*. 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 the previous OLA. Theoretically, 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*.

Unlike integers or relations, however, evolving data frame (edf) represents an *evolving* object (not a *state* of memory); thus, it is less obvious how we should understand/define the operations

that map an evolving object to another evolving object. We consider edf as an object consisting of a series of $K$ states, where each state contains a converging estimate for the final answer. Also, we consider an operation on edf as a map from a set of states to another set of states by appropriately modifying the information inside each state (i.e., estimates and metadata) depending on the types and the parameters of the operation.

To formalize those states and operations on them, we start with a few example data frames that represent different types of transformations each data frame may go through during OLA (§2.2), based on which we will formalize edf in the following sections (§3,§4,§5).

### 2.2 OLA Operations: Case Analysis

With the example in §1, 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 §3 and §4.

***Order-preserving Local Operation.*** 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 filters and maps, newly appearing rows in an input data frame do not affect the results of already processed rows. From the example in §1, L1 (read_csv), L3 (group-by on keys), L5 (filter), and L7 (join) belong to this category. See Fig 2a for illustration. Specifically, let df be an input data frame consisting of two partitions, i.e., df = [$\mathsf{df}_1$, $\mathsf{df}_2$], where "[]" indicates union/append. For such a *local operation* op, op(df) = [op($\mathsf{df}_1$), op($\mathsf{df}_2$)]; thus, unlike other cases described shortly, computing op($\mathsf{df}_2$) is independent from $\mathsf{df}_1$, 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($\mathsf{dfa}_1$, dfb), join($\mathsf{dfa}_2$, dfb)]; its physical plan may opt for different join algorithms such as progressive-merge [29] or hash joins.

***Shuffling Operation with Inference.*** If an input data frame is aggregated by a non-key attribute, e.g., lg_order_cust.sum(sum_qty, by=name) in §1, 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. From the example in §1, L8 (group-by on non-key attributes) belongs to this category. Fig 2b depicts the data flows of this case: the newly appearing rows in the input may affect an already produced output. Specifically, let df = [$\mathsf{df}_1$, $\mathsf{df}_2$]. For a *shuffling operation* op, op(df) = op($\mathsf{df}_1$) $\oplus$ op($\mathsf{df}_2$), where $\oplus$ indicates a key-based merge, which can be expressed as A $\oplus$ B = agg(union(A, B), by=key).[3]

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 converging estimates of the final answer, the raw sum values (without additional scaling) are also converging estimates of the output (§4.5).

[3]Merge operations are applicable to sum-like (or *mergeable*) operations, for which *addition* can be defined. Accordingly, avg() needs to be computed by separately computing sum() and count(). One notably hard case is count-distinct, for which we maintain exact sets (not HLL-based sketches [31]).

***Shuffling Operation without Inference.*** Operations like order-by and limit must consume the entire input, for which no special treatment can be applied to improve the quality of output. In these cases, upon a change of input, the output simply needs to be recomputed in its entirety, which, unlike Cases 1 and 2, cause inevitably redundant computation. From the example in §1, L9 (order-by and limit) belong to this category. For large-scale aggregation, however, these Case 3 operations typically appear in the latter stages to limit/sort the result for user consumption (e.g., bar charts); thus, their overhead is insignificant in the context of overall computations. Nevertheless, if a user's 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.

### 2.3 Required Properties

The case analysis in §2.2 reveals two types of changes: changes to attribute values (e.g., as aggregating more input rows) and cardinality growth (e.g., filtering input rows as they appear).

***Mutable Attributes.*** Let a *mutable attribute* be an attribute whose values may change whereas a *constant attribute* be an attribute whose values *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 DATA MODEL

This section describes evolving data frames (edf) from a user's perspective. Specifically, we describe its data model (§3.1), operations on it (§3.2), and current limitations (§3.3).

### 3.1 Evolving Data Frame

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

```
edf := t -> df  (0 <= t <= 1)
df := list((attr1, attr2, ..., attrM))
attr := constant_attr | mutable_attr
```

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.

***Properties for Closure.*** A valid edf must satisfy two properties, namely 2Cs. ***(1) Consistency***: All the `df` associated with an edf has the same schema; that is, its list of attributes remains constant over $t$. ***(2) Convergence***: The `df` associated with $t_2$ is a more accurate estimator of the exact answer compared to the `df` associated with $t_1$ ($t_1 < t_2$) while the `df` at $t = 1$ is the exact answer. In other words, *an operation on edf must produce an edf that ensures these two properties, which guarantees that edf is* closed *under those operations.*

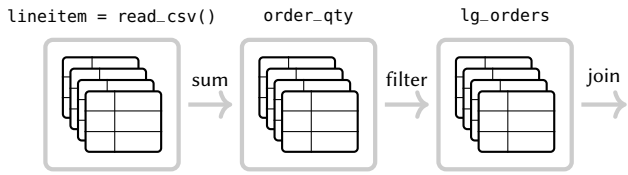



Fig. 3. User-view of evolving data frames (edf) and operations. Each edf expresses one or more states.

***Data Organization.*** 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 may 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, a partition may include the rows with orderkey between 1 and 10; then, other partitions must not contain the rows with orderkey=5. A clustering key may also be present for the edfs created as results of operations on other edfs, as we describe in §4. *A cluster of rows* refers to those rows present together in a partition.

***Accessing Values.*** As noted in §2, to represent an evolving data frame, an edf maintains *states*, 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 (by receiving eof). The final answer can be obtained by edf.get_final(), which may block until processing the entire data (if not already processed).

***Creating EDFs.*** There are two ways to create edfs. An edf can be created directly from a data source or an edf can be created as a result of the 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
```

where details of individual operations are described in §3.2. When creating an edf from a data source, a clustering key is obtained from metadata (§4.4). These operations and types of aggregations are sufficient to express all 22 TPC-H benchmark queries [6].

### 3.2 Operations on Evolving Data Frame

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

***Map.*** edf.map() resembles projection operations such as selecting a subset of attributes, creating derived attributes, etc. What's 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 edf2 such that edf2 = [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 [2], find_in_set [1]). Second, the approach easily enables efficient processing without additional logic for parallelizing/vectorizing row-wise functions.

***Join.*** edf.join(edf2, options) joins edf with edf2 as specified in its options, e.g., method (inner/left) and join keys. Depending on the join keys and clustering keys, Wake uses a different join method (i.e., hash or progressive-merge [29]). Specifically, if both edf and edf2 are clustered on their respective join keys, Wake performs a merge join; otherwise, Wake performs a hash join with edf as the *probe table* and edf2 as the *build table* (used for creating a hash table). If multiple joins are chained (e.g., edf.join(edf2).join(edf3)) and hash joins must be used, Wake effectively performs the right-deep join by constructing hash tables in parallel for edf2 and edf3, which is effective for star schema models [39].

***Aggregate.*** edf.agg(cols, by_attr) aggregates a group of rows (for each `by_attr`) where agg is one of the 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. §4 and §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 §4.

### 3.3 Limitation

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 (§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.

## 4 INTERNAL PROCESSING

Depending on the types of operations, our system (Wake) takes different approaches to update edf.

### 4.1 Properties of Evolving DataFrame

In addition to the schema described in §3.1, each edf maintains two additional properties, namely *progress* and *growth*, to characterize its evolution quantitatively.

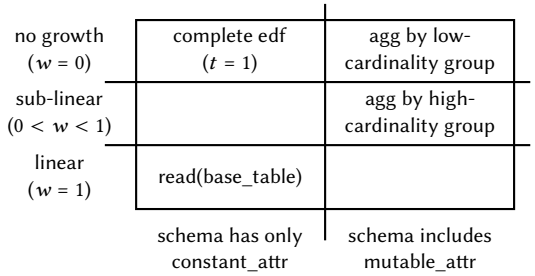



Fig. 4. edf types with examples categorized by degree of growth $w$ on Y-axis and attribute types on X-axis.

***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) tuples that must be processed to obtain the final answer; the total tuple count comes from metadata (§4.4). 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) is read/processed, $t$ is 1. If $t$ is 1, edf.is_final=True.

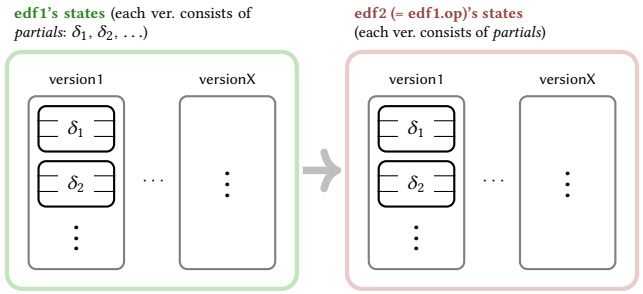
***Growth.*** *Growth* describes the growth of the current tuple count to forecast the final tuple count. Wake compactly models the growth as a monomial $ct^w$ using past observations. Fig 4 gives some examples with different $w$ values. Growth captures the local tuple count, while progress $t$ captures 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.

***Examples.*** 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 sub-linear growth. 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 $t = 1$. Fig 4 classifies the types of edf properties based on the degree of growth ($w$) and attribute types (constant/mutable). Its cells list a few examples that would result in edfs with such 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 appears). Accordingly, its schema includes mutable attributes, and $w$ is between 0 (no growth) and 1 (linear growth).

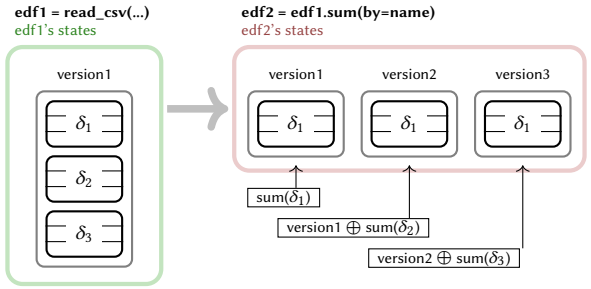
### 4.2 State Representation

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 edfs or other applications, whereas intrinsic states are used to incrementally maintain computed values prior to adjustments and/or estimations.

***Examples.*** Suppose we are counting the number of students by their home states. Let edf1 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 states $\alpha_1$ of edf1 becomes [[(id1,




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




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

Fig. 5. States-based representation of edf and operations on them (i.e., from edf1's extrinsic states to edf2's intrinsic states). A new state is (conceptually) defined each time a *partial* is added to the latest version or a new *version* is created. Creating a new edf (and its states) incurs low computational redundancy.

IL), (id2, IL), (id3, MI)]]. For edf1, its extrinsic states $\beta_1$ is identical to $\alpha_1$ because edf1—representing tuples from a base table—requires no adjustments. Let edf2 represent edf1.count(by=state); its intrinsic states $\alpha_2$ becomes [[(IL, 2), (MI, 1)]]. To express unbiased estimates, edf2's extrinsic states $\beta_2$ is scaled accordingly under the assumption that the unobserved (nine) partitions have the same distribution as the observed (first) partition; thus, $\beta_2$ 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 $\alpha_2$, we first aggregate the second list of $\beta_1$, temporarily obtaining [(IL, 1), (MI, 1)], which is merged into $\alpha_2$ using key-based sum ($\oplus$), as described in §2.2, finally obtaining $\alpha_2$ = [(IL, 3), (MI, 2)]. To obtain unbiased estimates from $\alpha_2$, we scale individual aggregate values considering the ratio between currently processed tuples and the total tuple count (i.e., 2:10), thereby obtaining edf2's extrinsic states $\beta_2$ = [(IL, 15), (MI, 10)].

Note that we have taken two different approaches in updating intrinsic states depending on edfs. For edf1, we have inserted new tuples, creating a longer list for $\alpha_1$; in contrast, for edf2, 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 5. Creating a new version means a complete refresh while appending partial(s) to each version (of an edf) means incremental updates.

Table 2. State transformation for each edf operation. GBI: growth-based inference.

| edf op | intrinsic repr. | merge ($\oplus$) | 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/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 | var/sum/count by key | avg/sum/sum by key | GBI |
| stddev | var/sum/count by key | avg/sum/sum by key | GBI |

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 5), 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 (§4.3). 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 edf1's intrinsic states $\alpha_1$ consist of two partials $\delta_1$ and $\delta_1$. To incrementally compute count-distinct (i.e., first using $\delta_1$ and then to update it using $\delta_1$), it is insufficient to have the number of unique values appearing in $\delta_1$ because count_distinct($\delta_1$) + count_distinct($\delta_2$) is *not* equal to count_distinct($\delta_1 \cup \delta_2$); we need to record all the individual unique values in $\delta_1$ to properly examine if the tuples in $\delta_2$ overlap with any of the values in $\delta_1$. 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 a 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 §5.

### 4.3 Operation: State Transformation

An operation in edf2 = edf1.op(...) is a state transformation process from edf1's extrinsic states to edf2's intrinsic states (which can then be used to produce edf2'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 $\alpha_2$ for each operation.

***Merge.*** To incrementally construct $\alpha_2$ 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 ($\oplus$) that can satisfy $\text{op}([\delta_1, \ldots, \delta_p]) = \text{op}(\delta_1) \oplus \ldots \oplus \text{op}(\delta_p)$. That is, given $\delta_1$, we can first compute $\text{op}(\delta_1)$; then, given $\delta_2$, we update the result by merging $\text{op}(\delta_2)$ into the previous result; this update operation continues for each partial.

For example, suppose we are computing $\text{avg}([\delta_1, \delta_2, \delta_3])$, 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 $\delta_1$, which is stored as an intrinsic state. Given the next partial ($\delta_2$), we (again) compute (count, sum_salary) for each state from $\delta_2$, then add these aggregates into the earlier results for each state, which is equal to directly computing (count, sum_salary) from a union of $\delta_1$ and $\delta_2$. Note that for each op, these intrinsic state representations and merge operations differ, which we summarize in Table 2.

***Primary Key.*** As described in §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 as a result of aggregation if the aggregation's grouping attributes are not identical to the clustering key itself.

***Other Properties.*** Besides attribute types, a new edf maintains 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 modifications. 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. We discuss this logic in §5.

### 4.4 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 corresponding to the tables that are being read. This is all the metadata that Wake requires from the underlying data, without requiring any other statistics. This metadata information is used for computing *progress* ($t$).

### 4.5 Closure of edf Properties

Wake's internal processing is designed to ensure 2Cs (§3.1) required to ensure the validity of all edfs through processing. Specifically, we satisfy (1) consistency and (2) convergence, as follows.

***Consistency.*** Every operation in §3.2 is a function mapping input edf(s) to an output edf with a fixed schema. Because the source of edf (`read` operation) always generates an edf with a fixed schema, all intermediate and final edfs have the same schema.

***Convergence.*** There are two types of convergence: (1) mutable attributes become more accurate, and (2) the key set (e.g., group-by attributes) converges. *Attribute convergence*: First of all, all the attribute values produced by Wake 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, $\text{E}[|\tilde{x}_n - x|] \leq \text{E}[|\tilde{x}_{n'} - x|]$ 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 [49]. Moreover, for mean-like aggregates (e.g., count, sum, avg, stddev,

var), we produce unbiased estimates; that is, $\mathrm{E}[\tilde{x}_n - x] = 0$. 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 [35, 82]. *Key-set convergence*: In approximate computing, a major source of non-existing keys is insufficient samples from the input data [18]. Nevertheless, under our framework, the key set converges to the true set because our operations are designed to produce the exact answers when the entire input data is observed.

## 5 AGGREGATE INFERENCE

Given an edf's intrinsic states, aggregate inference produces its extrinsic states. There are two 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 (§5.1) decomposes into two parts: cardinality estimator (§5.2) and aggregate estimators (§5.3).

### 5.1 Problem Decomposition

Wake formulates aggregate inference as an unbiased estimation problem. Using intrinsic states up until current progress $0 \le t \le 1$, aggregate inference aims to find per-cell unbiased estimators at final progress $T = 1$. Suppose the data frame has $m(t)$ groups, $X_i(t)$ denotes the $i$-th group cardinality (i.e. the number of tuples that have been aggregated into the $i$-th group) at progress $t$. Although many aggregate attributes may be present, aggregate inference focuses on each attribute at a time, referring to the aggregate values of the $i$-th group as $Y_i(t)$. Because unobserved partitions are unknown to Wake, $m(t)$, $X_i$, and $Y_i$ are stochastic processes over "time" $t$. We write the observed group cardinalities and aggregate values until progress $t$ in lower cases: $x_{i,:t}$ and $y_{i,:t}$ respectively. The desired unbiased estimator $\hat{y}_{i,:t}$ is the one such that:

$$\hat{y}_{i,:t} = \mathrm{E}\left[Y_i(T) \mid x_{i,:t}, y_{i,:t}\right] \tag{1}$$

Many known aggregate estimators rely on the current count $x_{i,t}$ and the final count $x_{i,T}$; however, the latter is not known at the current time. Instead, Wake computes an unbiased estimator of final group cardinality $\hat{x}_{i,:t}$ from group cardinalities so far $x_{i,:t}$ described in §5.2.

$$\hat{x}_{i,:t} = \mathrm{E}\left[X_i(T) \mid x_{i,:t}\right] \tag{2}$$

Using this estimator as well as the current cardinality and aggregate value, Wake then estimates the final aggregate value at $T$ by aggregate-aware estimators $f$ in §5.3.

$$\hat{y}_{i,:t} = \mathrm{E}\left[Y_i(T) \mid x_{i,:t}, y_{i,:t}\right] = f(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) \tag{3}$$

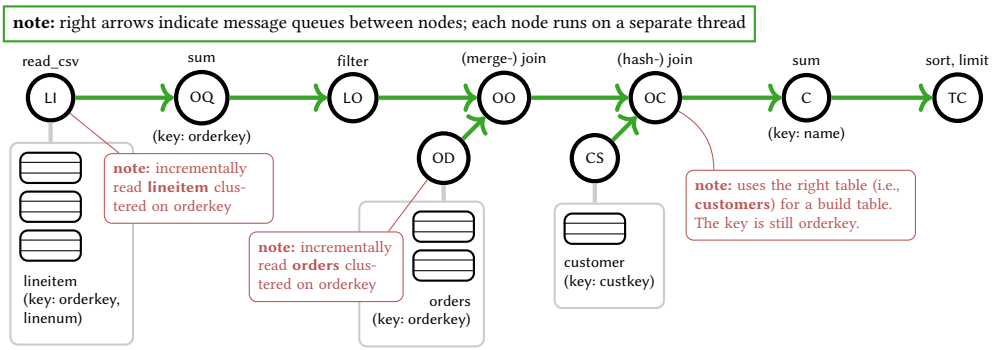
Therefore, Wake first estimates $\hat{x}_{i,:t}$ for all $i = 1, \ldots, m(t)$, and then estimates $\hat{y}_{i,:t}$. In a case of many aggregate attributes, Wake reuses $\hat{x}_{i,:t}$ to estimate each aggregate separately by applying the corresponding aggregate estimator. Finally, it collects all aggregate estimations into the output data frame filled with extrinsic states.

### 5.2 Cardinality Estimator

Wake models group cardinalities after *monomials* with a shared power, $\mathrm{E}[X_i(t)] \propto t^w$. The underlying reasoning is as follows. Wake assumes that the number of samples and the number of groups follow two hidden monomials, $\mathrm{E}[n(t)] \propto t^u$ and $\mathrm{E}[m(t)] \propto t^v$, 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 $w = u - v$. This modeling captures many scenarios in Deep OLA. For example, if the input data frame is a table reader, then Wake would expect the sample to grow linearly ($u = 1$). If the input is behind a cross join of

```
1  lineitem = read_csv('...')                          # LI
2  # item count for each order
3  order_qty = lineitem.sum(qty, by=orderkey)          # OQ
4  # select only the large orders
5  lg_orders = order_qty.filter(sum_qty > 300)         # LO
6  # find the customers with biggest order sizes
7  lg_order_cust = lg_orders.join(orders) \            # OO
8                    .join(customer)                   # OC
9  # select top-100 customers
10 qty_per_cust = lg_order_cust.sum(sum_qty, by=name)  # C
11 top_cust = qty_per_cust.sort(sum_qty, desc=True) \
12                 .limit(100)                         # TC
```

(a) Example data operation with edf




(b) Wake's internal execution plan

Fig. 6. Example data operations with edf (top) and Wake's internal representation for parallel process (bottom)

two tables, then Wake 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, Wake would see the number of groups grow 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, Wake does not need to estimate $\mathrm{E}[n(t)]$ nor $\mathrm{E}[m(t)]$, but only $\mathrm{E}[X_i(t)]$. Wake estimates final group cardinalities in two steps. First, it fits $w$ to the dataset consisting of average group cardinalities $\overline{x}_t = \frac{1}{m_t} \sum_{i=1}^{m_t} x_{i,t}$ for all observed $t$. Specifically, it fits the power $w$ as well as the coefficient $b$ in a logarithmic-transformed linear regression: $\mathrm{E}[\log \overline{x}_t] = \log b + w \log t$. Wake implements a streaming linear regression with $O(1)$ time/space complexities per observation. Finally, Wake fits each group's coefficient in $\mathrm{E}[X_i(t)|x_{i,:t}] = x_{i,t} = c_i t^w$ and predicts the final group cardinality with $T = 1$:

$$\hat{x}_{i,:t} = \mathrm{E}[X_i(T)|x_{i,:t}] = (x_{i,t}/t^w)\, T^w = x_{i,t} \;/\; t^w \tag{4}$$

### 5.3 Aggregate Estimators

Wake selects the aggregate estimator $f$ from the following set of estimators depending on the aggregation type. This set can be expanded together with existing estimators.

***Count.*** Use the estimated cardinality: $f_{\text{count}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = \hat{x}_{i,:t}$.

***Sum.*** Scale the summation: $f_{\text{sum}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = \frac{y_{i,t}}{x_{i,t}} \hat{x}_{i,:t}$.

***Weighted Avg.*** 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:

$$f_{\text{avg}}((y'_{i,:t}, y''_{i,:t}), x_{i,:t}, \hat{x}_{i,:t}) = \left(\frac{y'_{i,t}}{x_{i,t}}\hat{x}_{i,:t}\right) / \left(\frac{y''_{i,t}}{x_{i,t}}\hat{x}_{i,:t}\right) = y_{i,t} \tag{5}$$

***Count Distinct.*** Wake adopts a finite-population method-of-moment estimator [35] (in §4.1, denoted as $\hat{D}_{MM1}$). For brevity in this subsection, let us focus on the $i$-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 $y = y_{i,t}$. Wake computes $f_{cd}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = Y$ where $Y$ satisfies Equation (6).

$$y_{i,t} = Y(1 - h(\hat{x}_{i,:t}/Y)) \tag{6}$$

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

$$h(z) = \frac{\Gamma(\hat{x}_{i,:t} - z + 1)\,\Gamma(\hat{x}_{i,:t} - x_{i,t} + 1)}{\Gamma(\hat{x}_{i,:t} - x_{i,t} - z + 1)\,\Gamma(\hat{x}_{i,:t} + 1)} \tag{7}$$

***Order Statistics.*** Order statistics include min, max, median, quantiles, and $k$-th smallest/largest values. Currently, Wake simply outputs the latest value: $f_{\text{order}}(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t}) = y_{i,t}$ which provides a fairly accurate estimate for large $\hat{x}_{i,:t}$ at no computation cost.

### 5.4 Correctness

Given observations $(y_{i,:t}, x_{i,:t})$ up until current progress $t$, Lemma 1 and Lemma 2 together show that Wake's aggregate inference is unbiased under some conditions.

Lemma 1 (Unbiased Count). *$\hat{x}_{i,:t} = \mathrm{E}[X_i(T)|x_{i,:t}] = \frac{x_{i,t}}{t^w}$ is unbiased, if A) $w$ is unbiased and B) all operations produce a monomial or transform a monomially growing input(s) into a monomially growing output with respect to progress $t$.*

Lemma 2 (Unbiased Aggregation). *Given unbiased group cardinality estimate $\hat{x}_{i,:t} = \mathrm{E}[X_i(T)|x_{i,:t}]$, Wake's aggregate estimators produce unbiased estimates, possibly with additional conditions depending on aggregation type:* $\mathrm{E}\left[Y_i(T) \mid x_{i,:t}, y_{i,:t}\right] = f(y_{i,:t}, x_{i,:t}, \hat{x}_{i,:t})$.

Please find the proofs in our extended manuscript [75].

### 5.5 Alternatives

This section lists some of the alternative design choices we have considered but do not fit well with the broader picture of Wake.

***Probabilistic Cardinality Estimator.*** One could model the distribution $X_i(T)|x_{i,:t}$ (instead of the expectation $\mathrm{E}[X_i(T)|x_{i,:t}]$ in Wake) to express confidence. However, the evaluation would require computing the marginal expectation which may be expensive for many aggregate estimators. Moreover, which distribution to use is an open question to be investigated further.

***Other Cardinality Function Families.*** Different families of polynomials are attractive alternatives; however, one needs to know the set of orders *a priori* to efficiently fit their coefficients. Mixing exponential and logarithm could improve the accuracy in some cases but would also be more difficult to estimate. In contrast, affine functions are simple with many well-known estimation

algorithms, but they are only restricted to a specific growth pattern. Ultimately, monomial is the simplest and cheap to fit (in logarithmic scale) yet provides a wide range of growth curves.

***Order Statistics under Finite Population.*** Given PDF/CDF, there exists a density function of the $k$-th order statistic [28]. Given that, we could evaluate the expectation at $\hat{x}_{i,:t}$ numerically to acquire an unbiased estimate. A similar analysis is possible for discrete variables as well. However, this method has a prohibitive computational cost in general to reconstruct PDF/CDF, let alone evaluating the expectation. For example, if the reconstruction uses kernel density estimation (KDE) [68, 72] and empirical distribution function (eCDF), it would require $O(n(t))$ time and space to compute the density and hold all samples. Such a cost does not scale well and may straggle OLA progress with minimal accuracy gain.

## 6 CONFIDENCE INTERVAL FOR DEEP OLA

WAKE's mathematical concepts and implementation can be extended to offer confidence intervals in addition to mean estimates.

***Extended Definition.*** Wake maintains the "uncertainty" of all mutable attributes throughout processing in three steps: (1) it computes the uncertainty of initial mutable attributes, (2) propagates the uncertainty through edf operations, and (3) derives CIs from the final uncertainty. Specifically, we extend the edf definition (§3.1):

```
df_ci := (list((attr1, attr2, ..., attrM)), Σ)
```
(8)

where a *covariance matrix* $\Sigma$ captures the uncertainty with $\Sigma_{i,j}$ denoting the covariance between mutable attributes $\text{attr}_i$ and $\text{attr}_j$.

***Initial Variance.*** When mutable attributes first appear, Wake infer variances using the existing aggregation-specific variance estimators. Specifically, it measures the variance of cardinality power $\text{Var}(w)$ by calculating the variance of ordinary least square parameter [36]; sum and avg by applying central limit theorem [70]; count-distinct using Poissonization on empirical density function [15]; order statistics using bootstrapping [30]; and extreme order statistics (min/max) by fitting generalized extreme value distribution [46].

***Variance Propagation.*** Wake propagate $\Sigma$ through edf operations using a standard statistics technique: "propagation of uncertainty" [47]. For a differentiable mapping $v = f(u)$ and known covariance matrix $\Sigma^U$, Wake linearizes $f$ using first-order Taylor expansion and computes $\Sigma^V = J\Sigma^U J^{\mathrm{T}}$ where $J$ is the Jacobian matrix ($J_{i,k} = \partial f_i / \partial U_k$). Equation (9) expands this expression.

$$\Sigma^V_{i,j} = \sum_k \sum_l \Sigma^U_{k,l} \left(\partial f_i / \partial U_k\right) \left(\partial f_j / \partial U_l\right) \tag{9}$$

For instance, the covariance matrix propagates through `count` and `sum` as follows, respectively:

$$\text{Var}(f_{\text{count}}) = \text{Var}(\hat{x}_{i,:t}) = \left(\hat{x}_{i,:t} \ln(1/t)\right)^2 \text{Var}(w) \tag{10}$$

$$\text{Var}(f_{\text{sum}}) = 1/x^2_{i,t} \left(\text{Var}(y_{i,t})\hat{x}^2_{i,:t} + \text{Var}(\hat{x}_{i,:t}) y^2_{i,t}\right) \tag{11}$$

Please see our extended manuscript [75] for other operations (e.g., weighted avg, count distinct, order statistics, map/projection).

***Variance-based Confidence Interval.*** Finally, Wake derives a CI of an estimate $y$ from its variance $\sigma^2 = \text{Var}(y)$ based on Chevbyshev's inequality [78]. It outputs $[y - k\sigma, y + k\sigma]$ where $k = \sqrt{1/(1-\delta)}$ for confidence level $(1 - \delta)$. For example, $k \approx 4.5$ for 95% CI.

***Limitations.*** The above method applies to differentiable operations, which include the most we are interested in. Like the mean estimates, finite-variance uncertainty also assumes that the distribution of the observed data represents that of the unobserved, a fundamental premise of machine learning and statistical inference. CI calculation incurs time and memory overheads to compute Equation (9); however, these overheads are relatively small for TPC-H queries because only a small number of covariances are relevant.

***Alternatives.*** The variance propagation can be substituted with higher moments for enhanced accuracy; however, its time complexity (and runtime overheads) increases. Other alternatives, such as bootstrapping or the propagation of parametric distributions, either incur significant overhead or are applicable to limited operations.

## 7 IMPLEMENTATION

Wake's implementation in Rust can be majorly divided into two parts: (1) Query Service which lets the user build a query and (2) Execution Engine which executes the built query in an OLA manner.

### 7.1 Query Service

Users express a query as an execution graph composed of *nodes* representing different operations, and *edges* representing the data flow path between these nodes. A node has as many incoming edges (representing the operation's inputs) as the number of arguments appearing in an operation. For example, a join operation requires two incoming edges representing the edfs to be joined, whereas a basic filter operation requires one incoming edge. To support the edf operations described in §3.2, Wake implements different node types such as reader, merge-join, hash-join, aggregator, etc., allowing its users to express a large variety of queries. The edges are implemented using channels for sending a stream of messages across threads. The user can incrementally add nodes and edges to the query graph representing nested OLA ops.

The circles in Fig 6 represent the nodes and the green arrows represent the edges. The LI—read_csv—in the figure serves as the root, which passes fetched partitions to its subscriber (i.e., OQ), which continues as defined in the graph. The current leaf node (i.e., TC) represents the query output, which can be consumed by downstream applications (e.g., progressive visualization).

### 7.2 Execution Engine

The Execution Engine takes a Query Service and evaluates the query on a specified dataset generating a sequence of edf outputs. On specifying the input for each read_csv node, the execution engine starts the query execution. Each node operates in a separate thread, reading messages from its input channels. A received message consists of: (1) a shared pointer to a data frame and (2) metadata containing information on the progress of the query execution (§4). The node processes these messages, updates its intrinsic states using the metadata, and writes its extrinsic states along with the 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 messages on its input channels, the node sends an EOF message on its output channels and terminates its execution.

### 7.3 Discussion

***Query Optimization.*** As its first step, Wake introduces Deep OLA primitives without a declarative language. We plan to adopt existing optimization techniques such as predicate pushdown, join order optimization, etc., but will also investigate unique opportunities. For example, join algorithms (e.g., sort-merge or hash) affects not only the performance but the way that intermediate results

are delivered. If a subsequent group-by uses the same key as a join, we may opt for merge join for more interactive results even if a hash join can produce final results more quickly.

***OLA-Specific Optimizations.*** Wake's design includes OLA-specific optimizations such as (1) pipelined implementation of a query's operations, (2) sort-merge join when both the tables are partitioned on a common clustering key (e.g., lineitem and orders), (3) re-using the hash-table of right (build) tables for repeated hash-join, (4) shared pointers of data to reduce cloning costs, etc. These optimization help Wake produce exact answers as quickly as other systems designed for exact query processing.

***Intra-Query Parallelism.*** Wake benefits from both data pipelining and multi-threading, which are widely employed in data systems to reduce task completion time [56, 80]. Our extended report [75] studies the benefit of data pipelining. In the future, we will investigate Deep OLA-specific distributed processing.

## 8 EVALUATION

This section evaluates Wake against (conventional) exact data systems as well as OLA systems. Our experiments show the following:

- Wake's first estimate is 4.93× faster than the exact systems while being 1.3× slower in producing exact results. (§8.2)
- Wake's first estimates have a median error of 2.70%. Wake provides results with under 1% error 3.17× faster on average (upto 48.80×) than the best exact data system. (§8.3)
- Wake produces the results with less than 1% error, 1.92× times faster than state-of-the-art OLA systems. (§8.4)
- Wake's CIs comply with the chosen level of confidence but can be conservative towards the end of processing. (§8.5)
- Wake executes deep queries at expected time complexity. (§8.6)
- Wake's performance can be further improved by optimally choosing the partition sizes. (§8.7)

### 8.1 Experimental Setup

All the experiments are performed on a Standard D16ads v5 (Azure) machine with 16 vCPU(s) and 64 GB of memory. The TPC-H Benchmark is used for evaluation.

***Baselines.*** We employ 2 state-of-the-art OLA and 5 exact systems.

(1) ProgressiveDB [13]: A middleware-based OLA system for non-nested, join-free queries. We use the authors' implementation [5] (currently limited to a single table) and evaluate on TPC-H queries (Q1, Q6) for benchmark (Fig 9a).
(2) WanderJoin [49]: A random walk-based OLA system for non-nested, multi-join queries. We use the authors' implementation [7] with their modified queries (Q3, Q7, Q10) (Fig 9b).
(3) Presto [74]: Presto is a data warehouse designed for diverse data sources. We use its Hive connector on HDFS.
(4) Postgres: A popular RDBMS. We create appropriate FKs and indexes on the attributes according to the TPC-H schema.
(5) Polars [69]: Polars is a data frame library in Rust optimized with SIMD, Arrow [3], lock-free parallel hashing, etc.
(6) Vertica [48]: Vertica is a columnar storage-based analytical database. We use Vertica Data Warehouse on Azure.
(7) Actian Vector [93]: Actian Vector is a high-performance vectorized analytical database.

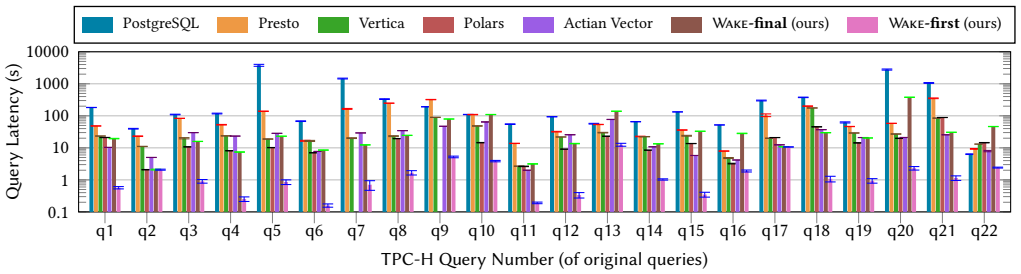



Fig. 7. Comparison of different baselines on TPC-H 100 GB dataset. The results are averaged across 10 runs.

***File Format.*** For Presto, Polars, and Wake, we use Parquet [4]. WanderJoin and ProgressiveDB are implemented on top of Postgres.

***Dataset.*** We use a scale-100 (100 GB) TPC-H dataset [6]. For Wake, the dataset is partitioned into 512 MB chunks, each of which is then converted to Apache Parquet format.

***Queries.*** We use the 22 TPC-H queries to evaluate the different systems and compare their performance. We employ TPC-H queries for two reasons. First, consistency with the existing work for accurate comparison. Second, many TPC-H queries can be considered *deep* since existing OLA methods cannot handle them due to nested `select` statements — except for Q1 and Q6. Relatedly, our comparison against existing OLA uses modified Q3, Q7, and Q10 (§8.4); however, for studying our system in §8.2 and §8.3, we use the original queries without simplifications. Finally, we evaluate Wake additionally with systematically generated deeper queries (§8.6).

***Metrics.*** The following metrics are computed:

- Final-Result Latency: The time taken to process the complete dataset and produce a correct final result.
- First-Estimate Latency: The time for the first estimate.
- Peak-Memory Usage: For in-memory Polars and Wake, we compute the peak-memory usage (i.e. maximum resident-set size).
- MAPE: Mean Absolute Percentage error is used to calculate the approximation error.
- Recall: For group-by queries, recall measures the fraction of final-result groups that were correctly produced.

### 8.2 Interactive Querying & Low Overhead

We compare Wake's latency against different exact query processing systems in terms of the time taken to produce first estimates and final exact results. Fig 7 shows the time taken by Wake to obtain the first and the final result. The number of intermediate results varies depending on the number of partitions of the tables the query uses. Wake produces first estimates 4.93× faster than Actian Vector's exact answers, 11.8× faster than Polars's exact answers, 21.81× faster than Vertica's exact answers, 78.3× faster than Presto's exact answers and 238.3× faster than Postgres's exact answers (all median). In terms of slowdown, measured as the ratio of Wake's final-result latency and other baseline's final-result latency, the median slowdown against Actian Vector is 5.3×, against Polars is 1.5×, against Vertica is 0.8×, against Presto is 0.3× and against Postgres is 0.1×, meaning Wake produces exact answers even faster than Vertica, Presto and Postgres. Wake—despite being an OLA system—produces exact results more quickly than Postgres in 20 queries, Presto in 17 queries, and Polars in 6 queries.

Moreover, Wake has low peak memory utilization. Polars runs out of memory (on the experiment machine) for queries Q7 and Q9, not able to generate results, whereas Wake successfully produces

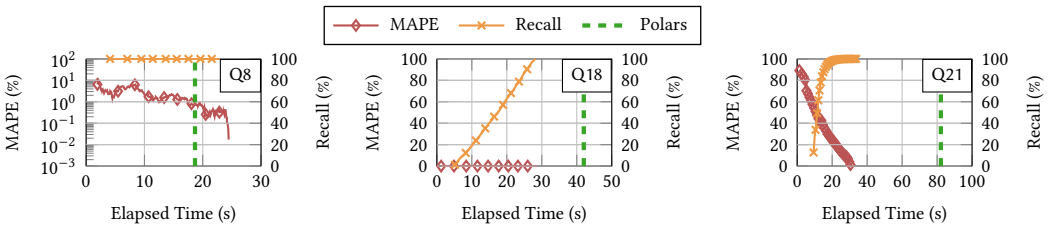



Fig. 8. Wake's approximation error measured in mean absolute percentage error (MAPE) and recall over time. Vertical lines represent completion time of exact methods. From left to right (Q8, Q18, Q21), Postgres completes in (332s, 376s, 1061s), while Presto completes in (247s, 200s, 352s) respectively.

exact query results. On average, Wake'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.

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 group-by on high-cardinality *c_custkey*) and Q20 (due to repeated filter on *partsupp*). For the first-estimate latencies, Wake has a median error of 2.70%. Wake provides estimated results with less than 1% MAPE, 3.17× faster than the final result time of the best baseline. WAKE's fast query performance benefits from our manual optimization such as predicate pushdown and careful choices of join methods (Fig 6); expectedly, we have observed poorer latencies with inferior query plans such as filtering after joins. This motivates our future work as discussed in §7.3.

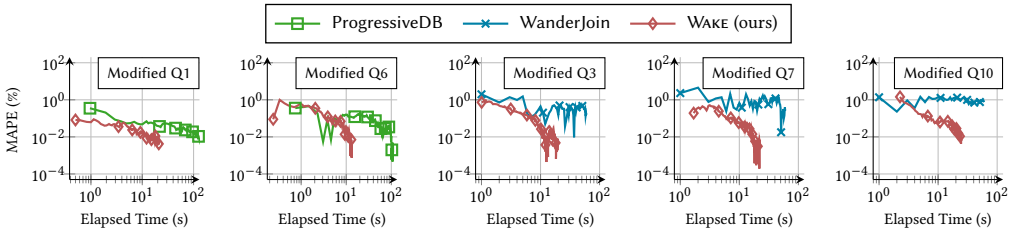
### 8.3 Wake's Approximation Error Analysis

In this experiment, we analyze approximate errors of Wake's OLA outputs (as it processes more data over time) in terms of MAPE and recall error. Fig 8 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 Wake observes more data while recalls reach 100% early on. Many queries in TPC-H fall into this category: Q1, Q4–Q9, Q12, Q14, Q17, Q19, and Q22. In particular, Q8 (Fig 8-left) involves a weighted average group-by aggregation over multiple joined tables. Wake is able to answer the first estimate at 1.9s with a 6.5% error. When Polars completes (at 19s), Wake has 0.87% error.

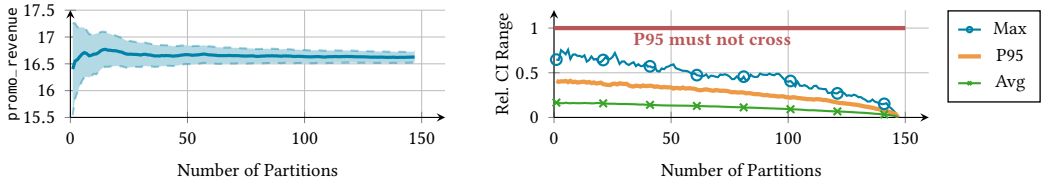
The queries in the second category involves clustering group-by keys; therefore, their aggregation values are exact (MAPE at 0%) while their recalls increase as Wake retrieve keys in different partitions. Q3, Q18 (Fig 8-right), and Q20 are examples: their recalls increase linearly as more keys are retrieved/observed.

Third, 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 (Fig 8-right) have quickly rising recall curves while their MAPE curves drop only linearly because their group-by keys are diverse, leading to a lower number of samples per group and so lower prediction power. While figures are omitted, 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 computes count over a high-cardinality non-clustering key (c_custkey), followed by an outer group-by over the inner count. Because the inner count changes




(a) vs ProgressiveDB (single-table Q1, Q6) (b) vs WanderJoin (modified Q3, Q7, Q10 [49])

Fig. 9. Comparison of approximation error over time against state-of-the-art OLA systems




(a) CI Convergence (b) CI Correctness

Fig. 10. Wake's 95% confidence interval on Q14 ($k \approx 4.5$).

over different partitions, the growth within outer groups can be non-monotonic, violating Wake's cardinality estimator and resulting in a relatively large MAPE.

### 8.4 Faster & More Accurate than Existing OLA

This section compares Wake with other OLA systems, ProgressiveDB and WanderJoin, on all of their supported sets of TPC-H queries: Q1, Q6 for ProgressiveDB, and Q3, Q7, Q10 for WanderJoin. Fig 9a shows the results from ProgressiveDB on Q1 and Q6. Although the initial estimates of ProgressiveDB and Wake are close, Wake converges 2.5× faster than ProgressiveDB to a less than 1% relative error. Fig 9b shows the comparison against WanderJoin. Although the errors of the first estimate are comparable, the convergence of Wake to a less than 1% relative error is 1.51× faster than WanderJoin. Moreover, Wake 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 Wake—converging to exact answers—is more desirable for end users.

### 8.5 Confidence interval correctness

We empirically verify Wake's CI derivation (§6) by executing Q14 with shuffled input partitions to simulate the inputs in unexpected orders. Q14 expresses a weighted average over a join of two tables with filters. Our computed confidence intervals converged toward the mean estimates as shown in Fig 10a. Moreover, we investigate the quality of those confidence intervals (Fig 10b), with *relative CI ranges*: the fraction of the actual absolute error over the size of the CI $|\hat{y} - y|/(k\sigma)$. Initially, our P95 relative CI range is around 0.4 as expected because our Chevbyshev-based CI assigns $k \approx 4.5$. Later, relative CI ranges decrease over partitions; while being overly conservative, they safely bound the true answers.

### 8.6 Performance on Synthetic Deep Queries

We study how Wake's internal processing scales with query complexity. We synthetically generate a 100-partition dataset with 100M rows of 11 integer columns. Ten of which are group-by columns

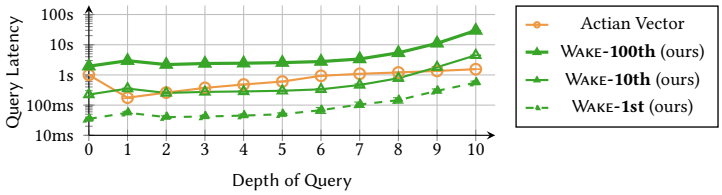



Fig. 11. Impact of query depth on its performance

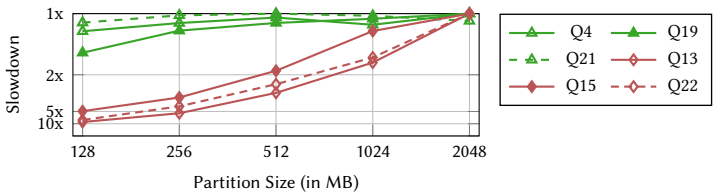



Fig. 12. Impact of partition size. Partition size is more influential in red queries (Q13, Q15, Q22) where the merge costs are higher than those in green ones (Q4, Q19, Q21).

having $4^{10}$ unique combinations. The synthetic query alternates between summation and maximum aggregations with a *query depth* $d = 0, \ldots, 10$. For example, if $d = 2$, the query is `df.max(x, by=(ci,cii)).sum(max_x, by=ci).sum(sum_max_x)`. Figure 11 shows Wake's latency to 1st, 10th, and 100th (final) results on a logarithmic y-axis. Across all depths, Wake takes similar times to process each partition and outputs the results at a regular pace. As the depth increases, Wake execution times scale with the primary group cardinality (i.e. $O(4^d)$ groups for depth $d$) because it needs to merge the new aggregate into the existing aggregate. In general, for $n$ rows, $B$ rows per partition, and depth $d$, Wake's time complexity is $O(4^d n/B + n)$ whereas non-OLA engine's complexity (e.g., Actian Vector) is expected to be $O(n)$.

### 8.7 Data Partition Size Matters

To understand the impact of partition sizes on overall query latencies, we evaluate Wake on scale-100 (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. Fig 12 shows the latencies of multiple TPC-H queries as a multiple of the best performance observed for that query across different partitions.

For queries with small merge operation overhead, the partition size expectedly does not affect the final-result latency. Some example queries are Q1, Q4, Q6, Q7, Q12, Q19 (group-by-agg has few groups), Q18, and Q21 (streamed on o_orderkey).

For queries with higher merge overhead (e.g., group-by with a large number of groups), the partition size makes a significant difference. Using larger partition sizes reduces the final-result latency because a smaller number of partitions leads to lower cumulative overhead. Some examples are Q13, Q15, Q16 (group-by-agg has large number of groups), and Q22 (pruning of c_custkey).

For less OLA-friendly queries (e.g., Q17), a larger partition size helps in reducing both first-result and final-result latencies. Hence, a suitable partition size depends on the query as well as the goal—be it either to minimize first-result latency or final-result latency. This work, however, does not investigate such an optimizer.

## 9 RELATED WORK

Approximate Query Processing (AQP) allows users to analyze large datasets interactively at a fraction of the costs of executing exact queries. Despite the years of research, it has been less successful in supporting deeply nested queries [19].

***Online Aggregation.*** Hellerstein et al. [38] first introduced the idea of OLA. Since then, various works [29, 34, 43, 49, 52] have built on top to increase the extent of queries supported, more focused on join queries. RippleJoin [34] progressively joins multiple tables, [52] improves online join algorithms for queries with low cardinality or a high number of groups. [29] provides OLA support for sort-based join algorithms. [42, 43] provides a scalable disk-based approach while providing better statistical bounds for sort-based join algorithms. WanderJoin [49] efficiently handles queries with multiple joins using indexes. Supporting nested aggregates and sub-queries has been limited in the literature. G-OLA [90] generalizes OLA to nested predicates by dividing the processed tuples into certain and uncertain sets based on running estimates. Some related works [84] also look at OLA in a distributed setting. [24, 60, 77] applies OLA to MapReduce.

***Incremental View Maintenance.*** Materialized views [9, 14, 20] provide improved query performance at the cost of additional storage and maintenance overhead. The base table changes require updating the materialized view using a delta query. Various techniques [10, 32, 33, 50, 59, 73, 87] have been proposed to perform incremental updates to materialized views. View maintenance and indexes [22, 23, 62] are fundamentally different from OLA since unlike OLA, it does not aim to estimate future results, which involves managing/propagating uncertainty over multiple operations.

***Approximate Query Processing.*** AQP includes synopses-based techniques [26] using samples [8, 11, 12, 17, 37, 63–66], wavelets [16], histograms [41, 71], sketches [25], etc. Recent ML-based works formulate AQP as a data-learning problem—through density estimators [40, 53], regression models [53], generative models [61, 79, 86]. More recently, query-aware generative models have been used to improve approximation error for low-cardinality queries [76, 92]. Challenges with these approaches include the high cost of model maintenance and re-training, the limited set of supported queries, and the difficulty in providing correctness bounds.

***Cardinality Estimation.*** Cardinality estimation is a fundamental problem in query optimization. Traditional approaches include using synopses like histograms, sketches, and samples [26, 67]. Recently, learning-based methods involving deep autoregressive models [88, 89], and ensemble-based methods [51] are gaining traction. [83] provides a thorough comparison. Any improvements in cardinality estimation can also improve Wake's accuracy.

## 10 CONCLUSION

This work takes a step toward Deep OLA (Online Aggregation) with a novel data model that is *closed* under set-oriented operations (e.g., map/filter/join/agg), thus enabling subsequent operations to previous OLA outputs. We show its viability through Wake—a Deep OLA system implemented in Rust. We have evaluated Wake on TPC-H (100 GB) by comparing it against state-of-the-art OLA engines and conventional data systems. Our experiments show that Wake provides first estimates 4.93× faster (median) than conventional systems' computing exact answers while offering a median 2.70% relative error. Moreover, Wake incurs small overhead (1.3× median slowdown) in producing exact answers. In fact, the pipelined implementation of different ops in Wake often provides faster total latencies than exact query processing engines for some queries. In the future, we aim to extend Wake to support a SQL-like declarative interface with automated query optimizations. We also aim to extend Wake to a distributed setup for higher scalability.

## ACKNOWLEDGMENTS

We thank anonymous reviewers for their feedback. This work is supported in part by Microsoft Azure.

## REFERENCES


[1] Accessed: 2022-10-01. MySQL 8.0 Reference - FIND_IN_SET. https://dev.mysql.com/doc/refman/8.0/en/string-functions.html#function_find-in-set.

[2] Accessed: 2022-10-01. MySQL 8.0 Reference - GROUP_CONCAT. https://dev.mysql.com/doc/refman/8.0/en/aggregate-functions.html#function_group-concat.

[3] Accessed: 2022-10-15. Apache Arrow. https://arrow.apache.org/.

[4] Accessed: 2022-10-15. Apache Parquet. https://parquet.apache.org/.

[5] Accessed: 2022-10-15. ProgressiveDB. https://github.com/DataManagementLab/progressiveDB.

[6] Accessed: 2022-10-15. TPC-H: Decision Support Benchmark. https://www.tpc.org/tpch/.

[7] Accessed: 2022-10-15. XDB: approXimate DataBase (XDB). https://github.com/InitialDLab/XDB.

[8] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. 2013. BlinkDB: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*. 29–42.

[9] Divyakant Agrawal, Amr El Abbadi, Ambuj Singh, and Tolga Yurek. 1997. Efficient view maintenance at data warehouses. *ACM SIGMOD Record* 26, 2 (1997), 417–427.

[10] Yanif Ahmad and Christoph Koch. 2009. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proceedings of the VLDB Endowment* 2, 2 (2009), 1566–1569.

[11] Brian Babcock, Surajit Chaudhuri, and Gautam Das. 2003. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 539–550.

[12] Johes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: practical privacy-preserving approximate query processing for data federations. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2691–2705.

[13] Lukas Berg, Tobias Ziegler, Carsten Binnig, and Uwe Röhm. 2019. ProgressiveDB: progressive data analytics as a middleware. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1814–1817.

[14] Jose A Blakeley, Per-Ake Larson, and Frank Wm Tompa. 1986. Efficiently updating materialized views. *ACM SIGMOD Record* 15, 2 (1986), 61–71.

[15] Leonid V. Bogachev, Alexander V. Gnedin, and Yuri V. Yakubovich. 2008. On the variance of the number of occupied boxes. *Advances in Applied Mathematics* 40, 4 (2008), 401–432. https://doi.org/10.1016/j.aam.2007.05.002

[16] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. 2001. Approximate query processing using wavelets. *The VLDB Journal* 10, 2 (2001), 199–223.

[17] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. 2007. Optimized stratified sampling for approximate query processing. *ACM Transactions on Database Systems (TODS)* 32, 2 (2007), 9–es.

[18] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 511–519.

[19] Surajit Chaudhuri, Bolin Ding, and Srikanth Kandula. 2017. Approximate Query Processing: No Silver Bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 511–519. https://doi.org/10.1145/3035918.3056097

[20] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 190–200.

[21] Shimin Chen, Phillip B Gibbons, and Suman Nath. 2010. Pr-join: a non-blocking join achieving higher early result rate with statistical guarantees. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 147–158.

[22] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2022. Automatically Finding Optimal Index Structure. *arXiv preprint arXiv:2208.03823* (2022).

[23] Supawit Chockchowwat, Chaitanya Sood, and Yongjoo Park. 2022. Airphant: Cloud-oriented Document Indexing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1368–1381.

[24] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M Hellerstein, John Gerth, Justin Talbot, Khaled Elmeleegy, and Russell Sears. 2010. Online aggregation and continuous query support in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 1115–1118.

[25] Graham Cormode. 2011. Sketch techniques for approximate query processing. *Foundations and Trends in Databases. NOW publishers* (2011), 15.

[26] Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. 2011. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases* 4, 1–3 (2011), 1–294.

[27] Andrew Crotty, Alex Galakatos, Emanuel Zgraggen, Carsten Binnig, and Tim Kraska. 2016. The case for interactive data exploration accelerators (IDEAs). In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. 1–6.

[28] Herbert A David and Haikady N Nagaraja. 2004. *Order statistics*. John Wiley & Sons.

[29] Jens-Peter Dittrich, Bernhard Seeger, David Scot Taylor, and Peter Widmayer. 2002. Progressive merge join: a generic and non-blocking sort-based join algorithm. In *Proceedings of the 28th international conference on Very Large Data*

*Bases*. 299–310.
[30] Bradley Efron. 1979. Bootstrap Methods: Another Look at the Jackknife. *The Annals of Statistics* 7, 1 (1979), 1 – 26. https://doi.org/10.1214/aos/1176344552
[31] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 2007. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*. Discrete Mathematics and Theoretical Computer Science, 137–156.
[32] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *ACM SIGMOD Record* 30, 2 (2001), 331–342.
[33] Ashish Gupta, Inderpal Singh Mumick, and Venkatramanan Siva Subrahmanian. 1993. Maintaining views incrementally. *ACM SIGMOD Record* 22, 2 (1993), 157–166.
[34] 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*. 287–298.
[35] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. 1995. Sampling-Based Estimation of the Number of Distinct Values of an Attribute. In *VLDB*. Morgan Kaufmann, 311–322.
[36] Fumio Hayashi. 2000. *Econometrics*. Princeton University Press. 27–32 pages.
[37] Wen He, Yongjoo Park, Idris Hanafi, Jacob Yatvitskiy, and Barzan Mozafari. 2018. Demonstration of VerdictDB, the platform-independent AQP system. In *Proceedings of the 2018 International Conference on Management of Data*. 1665–1668.
[38] Joseph M Hellerstein, Peter J Haas, and Helen J Wang. 1997. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. 171–182.
[39] Ian Hellström. Accessed: 2022-10-01. Oracle SQL & PL/SQL Optimization for Developers. https://oracle.readthedocs.io/en/latest/sql/joins/hash-join.html.
[40] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2019. Deepdb: Learn from data, not from queries! *arXiv preprint arXiv:1909.00607* (2019).
[41] Yannis E Ioannidis and Viswanath Poosala. 1999. Histogram-based approximation of set-valued query-answers. In *VLDB*, Vol. 99. 174–185.
[42] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. 2008. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)* 33, 4 (2008), 1–54.
[43] Christopher Jermaine, Alin Dobra, Subramanian Arumugam, Shantanu Joshi, and Abhijit Pol. 2006. The sort-merge-shrink join. *ACM Transactions on Database Systems (TODS)* 31, 4 (2006), 1382–1416.
[44] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. 2016. QuickR: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 international conference on management of data*. 631–646.
[45] Albert Kim, Eric Blais, Aditya Parameswaran, Piotr Indyk, Sam Madden, and Ronitt Rubinfeld. 2015. Rapid sampling for visualizations with ordering guarantees. In *Proceedings of the vldb endowment international conference on very large data bases*, Vol. 8. NIH Public Access, 521.
[46] Samuel Kotz and Saralees Nadarajah. 2000. *Extreme Value Distributions*. PUBLISHED BY IMPERIAL COLLEGE PRESS AND DISTRIBUTED BY WORLD SCIENTIFIC PUBLISHING CO. https://doi.org/10.1142/p191 arXiv:https://www.worldscientific.com/doi/pdf/10.1142/p191
[47] Harry H. Ku. 2010. Notes on the Use of Propagation of Error Formulas. In *Journal of Research of the National Bureau of Standards, Section C: Engineering and Instrumentation*, Vol. 2.
[48] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. 2012. The vertica analytic database: C-store 7 years later. *arXiv preprint arXiv:1208.4173* (2012).
[49] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data*. 615–629.
[50] Zhaoheng Li, Xinyu Pi, and Yongjoo Park. 2023. S/C: Speeding up Data Materialization with Bounded Memory. In *2023 IEEE 39th international conference on data engineering (ICDE)*. IEEE.
[51] Jie Liu, Wenqian Dong, Qingqing Zhou, and Dong Li. 2021. Fauce: fast and accurate deep ensembles with uncertainty for cardinality estimation. *Proceedings of the VLDB Endowment* 14, 11 (2021), 1950–1963.
[52] 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*. 252–262.
[53] Qingzhi Ma and Peter Triantafillou. 2019. Dbest: Revisiting approximate query processing engines with machine learning models. In *Proceedings of the 2019 International Conference on Management of Data*. 1553–1570.
[54] Wes McKinney et al. 2011. pandas: a foundational Python library for data analysis and statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.
[55] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential Dataflow.. In *CIDR*.

[56] John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, et al. 2015. S-Store: Streaming Meets Transaction Processing. *Proceedings of the VLDB Endowment* 8, 13 (2015).

[57] Mohamed F Mokbel, Ming Lu, and Walid G Aref. 2004. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *Proceedings. 20th International Conference on Data Engineering*. IEEE, 251–262.

[58] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 439–455.

[59] Milos Nikolic, Mohammed Elseidy, and Christoph Koch. 2014. LINVIEW: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 253–264.

[60] Niketan Pansare, Vinayak Borkar, Chris Jermaine, and Tyson Condie. 2011. Online aggregation for large mapreduce jobs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1135–1145.

[61] Noseong Park, Mahmoud Mohammadi, Kshitij Gorde, Sushil Jajodia, Hongkyu Park, and Youngmin Kim. 2018. Data synthesis based on generative adversarial networks. *arXiv preprint arXiv:1806.03384* (2018).

[62] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2015. Neighbor-Sensitive Hashing. *Proceedings of the VLDB Endowment* 9, 3 (2015), 144–155.

[63] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. 2016. Visualization-aware sampling for very large databases. In *2016 IEEE 32nd international conference on data engineering (ICDE)*. IEEE, 755–766.

[64] Yongjoo Park, Barzan Mozafari, Joseph Sorenson, and Junhao Wang. 2018. VerdictDB: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data*. 1461–1476.

[65] Yongjoo Park, Jingyi Qing, Xiaoyang Shen, and Barzan Mozafari. 2019. BlinkML: Efficient maximum likelihood estimation with probabilistic guarantees. In *Proceedings of the 2019 International Conference on Management of Data*. 1135–1152.

[66] Yongjoo Park, Ahmad Shahab Tajik, Michael Cafarella, and Barzan Mozafari. 2017. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 587–602.

[67] Yongjoo Park, Shucheng Zhong, and Barzan Mozafari. 2020. QuickSel: Quick selectivity learning with mixture models. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1017–1033.

[68] Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The annals of mathematical statistics* 33, 3 (1962), 1065–1076.

[69] pola rs. Accessed: 2022-10-14. Polars: Lightning-fast DataFrame library for Rust and Python. https://www.pola.rs/.

[70] Georg Pólya. 1920. Über den zentralen Grenzwertsatz der Wahrscheinlichkeitsrechnung und das Momentenproblem. *Mathematische Zeitschrift* 8 (1920), 171–181.

[71] Viswanath Poosala, Venkatesh Ganti, and Yannis E. Ioannidis. 1999. Approximate query answering using histograms. *IEEE Data Eng. Bull.* 22, 4 (1999), 5–14.

[72] Murray Rosenblatt. 1956. Remarks on some nonparametric estimates of a density function. *The annals of mathematical statistics* (1956), 832–837.

[73] Kenneth Salem, Kevin Beyer, Bruce Lindsay, and Roberta Cochrane. 2000. How to roll a join: Asynchronous incremental view maintenance. *ACM SIGMOD Record* 29, 2 (2000), 129–140.

[74] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[75] Nikhil Sheoran, Supawit Chockchowwat, Arav Chheda, Suwen Wang, Riya Verma, and Yongjoo Park. 2022. A Step Toward Deep Online Aggregation (Extended Version). *arXiv preprint arXiv:2303.04103* (2022).

[76] Nikhil Sheoran, Subrata Mitra, Vibhor Porwal, Siddharth Ghetia, Jatin Varshney, Tung Mai, Anup Rao, and Vikas Maddukuri. 2022. Conditional Generative Model Based Predicate-Aware Query Approximation. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 8 (Jun. 2022), 8259–8266. https://doi.org/10.1609/aaai.v36i8.20800

[77] Yingjie Shi, Xiaofeng Meng, Fusheng Wang, and Yantao Gan. 2012. You Can Stop Early with COLA: Online Processing of Aggregate Queries in the Cloud. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management* (Maui, Hawaii, USA) *(CIKM '12)*. Association for Computing Machinery, New York, NY, USA, 1223–1232. https://doi.org/10.1145/2396761.2398423

[78] P. Tchébychef. 1867. Des valeurs moyennes (Traduction du russe, N. de Khanikof. *Journal de Mathématiques Pures et Appliquées* (1867), 177–184. http://eudml.org/doc/234989

[79] Saravanan Thirumuruganathan, Shohedul Hasan, Nick Koudas, and Gautam Das. 2020. Approximate query processing for data exploration using deep generative models. In *2020 IEEE 36th international conference on data engineering (ICDE)*. IEEE, 1309–1320.

[80] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. 2014. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD*

*international conference on Management of data*. 147–156.
[81] Tolga Urhan and Michael J Franklin. 2000. XJoin: A Reactively-Scheduled Pipelined Join Operator. *Bulletin of the Technical Committee on* (2000), 27.
[82] A. W. van der Vaart. 1998. *Asymptotic Statistics*. Cambridge University Press. https://doi.org/10.1017/CBO9780511802256
[83] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2020. Are we ready for learned cardinality estimation? *arXiv preprint arXiv:2012.06743* (2020).
[84] Sai Wu, Shouxu Jiang, Beng Chin Ooi, and Kian-Lee Tan. 2009. Distributed online aggregations. *Proceedings of the VLDB Endowment* 2, 1 (2009), 443–454.
[85] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. 2010. Continuous sampling for online aggregation over multiple queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 651–662.
[86] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni. 2019. Modeling Tabular data using Conditional GAN. In *Advances in Neural Information Processing Systems*.
[87] Jian Yang, Kamalakar Karlapalem, and Qing Li. 1997. Algorithms for materialized view design in data warehousing environment. In *VLDB*, Vol. 97. 136–145.
[88] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: one cardinality estimator for all tables. *arXiv preprint arXiv:2006.08109* (2020).
[89] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).
[90] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. 2015. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 913–918.
[91] Kai Zeng, Sameer Agarwal, and Ion Stoica. 2016. IOLAP: Managing uncertainty for efficient incremental OLAP. In *Proceedings of the 2016 international conference on management of data*. 1347–1361.
[92] Meifan Zhang and Hongzhi Wang. 2021. Approximate query processing for group-by queries based on conditional generative models. *arXiv preprint arXiv:2101.02914* (2021).
[93] Marcin Zukowski, Mark Van de Wiel, and Peter Boncz. 2012. Vectorwise: A vectorized analytical DBMS. In *2012 IEEE 28th International Conference on Data Engineering*. IEEE, 1349–1350.

Received October 2022; revised January 2023; accepted February 2023