Noname manuscript No.
(will be inserted by the editor)

PF-OLA: A High-Performance Framework for Parallel Online Aggregation

Chengjie Qin · Florin Rusu

Received: date / Accepted: date

Abstract Online aggregation provides estimates to the final result of a computation during the actual processing. The user can stop the computation as soon as the estimate is accurate enough, typically early in the execution. This allows for the interactive data exploration of the largest datasets.

In this paper we introduce the first framework for parallel online aggregation in which the estimation virtually does not incur any overhead on top of the actual execution. We define a generic interface to express any estimation model that abstracts completely the execution details. We design a novel estimator specifically targeted at parallel online aggregation. When executed by the framework over a massive 8TB TPC-H instance, the estimator provides accurate confidence bounds early in the execution even when the cardinality of the final result is seven orders of magnitude smaller than the dataset size and without incurring overhead.

Keywords sampling · online estimation · parallel processing · approximate algorithms · user-defined aggregates (UDA)

1 Introduction

Interactive data exploration is a prerequisite in model design. It requires the analyst to execute a series of exploratory queries in order to find patterns or relationships in the data. In the Big Data context, it is likely that the entire process is time-consuming

Chengjie Qin
University of California, Merced
5200 N Lake Road
Merced, CA 95343
E-mail: cqin3@ucmerced.edu

Florin Rusu
University of California, Merced
5200 N Lake Road
Merced, CA 95343
E-mail: frusu@ucmerced.edu

even for the fastest parallel database systems given the size of the data and the sequential nature of exploration—the next query to be asked is always dependent on the previous. Online aggregation [18,19] aims at reducing the duration of the exploration process by allowing the analyst to rule out the non-informative queries early in the execution. To make this possible, an estimate to the final result of the query with progressively narrower confidence bounds is continuously returned to the analyst. When the confidence bounds become tight enough – typically early in the processing – the analyst can decide to stop the execution and focus on the next query.

Although introduced in the late nineties, it is hard to find any commercial system that supports online aggregation even today [12]. In our opinion, there are multiple reasons that hindered adoption given that multiple academic prototypes [6,31] proved the feasibility of the approach. The first argument underlines the negative effect online aggregation has on normal query execution, with increases of at least 30% being common [13,17]. This is regarded as unacceptable in the commercial world who focus instead on improving the performance of traditional database systems. The second argument addresses the lack of a unified approach to express estimation models. The authors went multiple times through the process of designing a new estimation model in previous projects [13]. Each time, major changes to the overall system architecture and a significant amount of implementation were required. These obstructed the ultimate goal of designing a better estimation model. The final argument against online aggregation adoption we mention is the requirement to re-implement the data processing system from ground up in order to support estimation.

PF-OLA (**P**arallel **F**ramework for **O**n**L**ine **A**ggregation) overpasses these limitations and brings online aggregation closer to broader adoption, especially for Big Data interactive exploration. PF-OLA is a shared nothing parallel system executing complex analytical queries over terabytes of data very efficiently. Estimates and corresponding confidence bounds for the query result are computed during the entire query execution process without incurring any noticeable overhead. Thus, a user executing a long-running query starts getting estimates with provable guarantees almost instantly after query processing starts. As the query progresses, the width of the confidence bounds shrinks progressively, converging to the true result when the query is complete. As far as we know, PF-OLA is the first system that incurs virtually no overhead on top of the query execution time corresponding to the non-interactive execution. This is due to the extensive use of parallelism at all levels of the system – including storage, inside a single node, and across all the nodes – and to a judicious overlapping of query execution and estimation. At the same time, neither the estimator accuracy nor the convergence rate are negatively impacted, but rather the convergence receives a significant boost from the parallel discovery of result tuples. This results in fast and accurate estimation even for highly selective queries with very low result cardinality – the needle in the haystack problem – or in the case of skewed data.

A second aspect that differentiates PF-OLA from other online aggregation systems is the unified approach to express estimation models. In PF-OLA, the estimation is closely intertwined with the actual computation, with both clearly separated from query execution. Any computation is expressed as a User-Defined Aggregate (UDA) [32]. The user is responsible for implementing a standard interface while the

framework takes care of all the issues related to parallel execution and data flow. Adding online estimation to the computation is a matter of extending the UDA state and interface with constructs corresponding to the estimator. In order to apply a different estimation model to a computation, a corresponding UDA has to be implemented. No changes have to be made to the implementation of the computation or to the implementation of the framework. This provides tremendous flexibility in designing a variety of estimation models.

To verify the expressiveness of the framework and test the performance, we design an asynchronous sampling estimator specifically targeted at parallel online aggregation. The estimator is defined over multiple data partitions which can be independently sampled in parallel. This allows for accurate estimates to be computed even when there is considerable variation between the sampling rate across partitions. We analyze the properties of the estimator and compare it with two other sampling-based estimators proposed for parallel online aggregation—a synchronous estimator [34] and a stratified sampling estimator [24]. All these estimators are expressed using the extended UDA interface and executed without any changes to the framework, thus proving the generality of our approach.

The complete system re-implementation is avoided in PF-OLA through the use of the generic UDA mechanism to express both computation as well as estimation. The framework defines only the execution strategy without imposing any limitation on the actual computation. As long as the user-provided computation and estimation model can be expressed using the generic mechanism, there is no need to change the PF-OLA implementation. And, as shown in [30, 14], the complexity of the tasks that can be expressed with the UDA mechanism ranges from simple and group-by aggregations to clustering and convex optimization problems applied in machine learning.

Our main contributions can be summarized as follows:

- We design the first framework for parallel online aggregation that incurs virtually no overhead on top of the actual execution. Estimates and corresponding confidence bounds are continuously computed based on samples extracted from data during the entire processing.
- We define a generic interface to express estimation models by extending the well-known UDA mechanism. The user is required to represent the model using a pre-defined set of methods while the framework handles all the execution details in a parallel environment.
- We implement the online aggregation framework in a highly-parallel processing system for the execution of arbitrary jobs. The result is an extremely efficient prototype for Big Data analytics with support for online aggregation.
- We propose a novel asynchronous sampling estimator for parallel online aggregation that we implement and execute inside the framework. We provide statistical analysis, verify the correctness, and show the superior performance when compared with two other estimators previously proposed in the literature.
- We run an extensive set of experiments to benchmark the estimator. When executed by the framework over a massive 8TB TPC-H instance [2], the estimator provides accurate confidence bounds early in the execution even when the cardinality of the result is seven orders of magnitude smaller than the dataset size or when data are skewed without incurring any noteworthy overhead on top of the

normal execution. Moreover, the estimator exhibits high resilience in the wake of processing node delays and failures.

Roadmap. In the remainder of the paper, we first introduce a set of preliminary concepts in Section 2. Parallel online aggregation is formalized in Section 3 which contains a detailed presentation of our proposed estimator and a thorough comparison with existent estimators. The material in these two sections is based on our previous analysis of sampling estimators for parallel online aggregation [27]. The design of the framework and the implementation details are discussed in Section 4. Example estimators and their implementation in PF-OLA are presented in Section 5, while Section 6 contains the empirical evaluation of the framework and of the proposed estimator. Related work is discussed in Section 7. We conclude by summarizing the main findings of this paper and providing future directions in Section 8.

2 Preliminaries

We consider aggregate computation in a parallel cluster environment consisting of multiple processing nodes. Each processing node has a multi-core processor consisting of one or more CPUs, thus introducing an additional level of parallelism. Data are partitioned into fixed size chunks that are stored across the processing nodes. Parallel aggregation is supported by processing multiple chunks at the same time both across nodes as well as across the cores inside a node.

We consider general SELECT-PROJECT-JOIN (SPJ) queries having the following SQL form [13]:

$$\begin{aligned} &\texttt{SELECT SUM(f(}t_1 \bullet t_2\texttt{))} \\ &\texttt{FROM TABLE}_1\texttt{ AS }t_1\texttt{, TABLE}_2\texttt{ AS }t_2 \\ &\texttt{WHERE P(}t_1 \bullet t_2\texttt{)} \end{aligned} \tag{1}$$

where $\bullet$ is the concatenation operator, f is an arbitrary *associative decomposable aggregate function* [30] over the tuple created by concatenating t_1 and t_2, and P is some boolean predicate that can embed selection and join predicates. The class of associative decomposable aggregate functions, i.e., functions that are associative and commutative, is fairly extensive and includes the majority of standard SQL aggregate functions. Associative decomposable aggregates allow for the maximum degree of parallelism in their evaluation since the computation is independent of the order in which data inside a chunk are processed as well as of the order of the chunks, while partial aggregates computed over different chunks can be combined together straightforwardly. While the paper does not explicitly discuss aggregate functions other than SUM, functions such as COUNT, AVERAGE, STD DEV, and VARIANCE can all be handled easily—they are all associative decomposable. For example, COUNT is a special case of SUM where $f(\cdot) = 1$ for any tuple, while AVERAGE can be computed as the ratio of SUM and COUNT.

GROUP BY queries can also be handled using the methods in this paper by simply treating each group as a separate query and running all queries simultaneously; then

all of the estimates are presented to the user. For each group, a version of `P` is used that accepts only tuples from that particular group.

2.1 Parallel Aggregation

Aggregate evaluation takes two forms in parallel databases. They differ in how the partial aggregates computed for each chunk are combined together. In the centralized approach, all the partial aggregates are sent to a common node – the coordinator – that is further aggregating them to produce the final result. As an intermediate step, local aggregates can be first combined together and only then sent to the coordinator. In the parallel approach, the nodes are first organized into an aggregation tree. Each node is responsible for aggregating its local data and the data of its children. The process is executed level by level starting from the leaves, with the final result computed at the root of the tree. The benefit of the parallel approach is that it also parallelizes the aggregation of the partial results across all the nodes rather than burdening a single node (with data and computation). The drawback is that in the case of a node failure it is likely that more data are lost. Notice that these techniques are equally applicable inside a processing node, at the level of a multi-core processor.

PF-OLA supports both centralized and parallel aggregation at the level of the entire cluster as well as inside each node. The strategy to be applied is determined dynamically for each query. Moreover, when online aggregation is executed simultaneously with the normal query execution, the aggregation strategy is chosen individually for each of the tasks. Thus, for example, it is possible to have the query executed with parallel aggregation, while the estimation is centralized.

2.2 Online Aggregation

The idea in online aggregation is to compute only an estimate of the aggregate result based on a sample of the data [18]. In order to provide any useful information though, the estimate is required to be accurate and statistically significant. Different from one-time estimation [12, 15] that might produce very inaccurate estimates for arbitrary queries, online aggregation is an iterative process in which a series of estimators with improving accuracy are generated. This is accomplished by including more data in estimation, i.e., increasing the sample size, from one iteration to another. The end-user can decide to run a subsequent iteration based on the accuracy of the estimator. Although the time to execute the entire process is expected to be much shorter than computing the aggregate over the entire dataset, this is not guaranteed, especially when the number of iterations is large. Other issues with *iterative online aggregation* [34, 23] regard the choice of the sample size at each iteration and reusing the work done from one iteration to the following.

An alternative that avoids these problems altogether is to completely *overlap query processing with estimation* [31, 25]. As more data are processed towards computing the final aggregate, the accuracy of the estimator improves accordingly. For this to be true though, data are required to be processed in a statistically meaningful

order, i.e., random order, to allow for the definition and analysis of the estimator. This is typically realized by randomizing data during the loading process. The drawback of the overlapped approach is that the same query is essentially executed twice—once towards the final aggregate and once for computing the estimator. As a result, the total execution time in the overlapped case is expected to be higher when compared to the time it takes to execute each task separately.

PF-OLA is designed as an online aggregation system which overlaps query execution with estimation. The motivation for this choice is the ever increasing number of cores available on modern CPUs. Since I/O is the bottleneck in database processing, the additional computation power is not utilized unless concurrent tasks are found and executed. Given that the estimation process requires access to the same data as normal processing, estimation is a natural candidate for overlapped execution. While it is straightforward to execute these two processes concurrently, the challenge is how to realize this such that the normal query execution time is not increased at all—this should always be possible as long as there are non-utilized cores on the processing node. We show how PF-OLA achieves this goal by carefully scheduling access to shared data across the two tasks.

3 Parallel Online Aggregation

An online aggregation system provides estimates and confidence bounds during the entire query execution process. As more data are processed, the accuracy of the estimator increases while the confidence bounds shrink progressively, converging to the actual query result when the entire data have been processed. There are multiple aspects that have to be considered in the design of a parallel online aggregation system. First, a mechanism that allows for the computation of partial aggregates has to be devised. Second, a parallel sampling strategy to extract samples from data over which partial aggregates are computed has to be designed. Each sampling strategy leads to the definition of an estimator for the query result, estimator that has to be analyzed in order to derive confidence bounds. We discuss in details each of these aspects for the overlapped online aggregation approach in this section. Then, in Section 4, we show how everything is implemented in the PF-OLA framework.

3.1 Partial Aggregation

The first requirement in any online aggregation system is a mechanism to compute partial aggregates over some portion of the data. Partial aggregates are typically a superset of the query result since they have to contain additional data required for estimation. The partial aggregation mechanism can take two forms. We can fix the subset of the data used in partial aggregation and execute a normal query. Or we can interfere with aggregate computation over the entire dataset to extract partial results before the computation is completed. The first alternative corresponds to iterative online aggregation, while the second to overlapped execution.

Partial aggregation in a parallel setting raises some interesting questions. For iterative online aggregation, the size and location of the data subset used to compute

the partial aggregate have to be determined. It is common practice to take the same amount of data from each node in order to achieve load balancing. Or to have each node process a subset proportional to its data as a fraction from the entire dataset. Notice though that it is not necessary to take data from all the nodes. In the extreme case, the subset considered for partial aggregation can be taken from a single node. Once the data subset at each node is determined, parallel aggregation proceeds normally, using either the centralized or parallel strategy. In the case of overlapped execution, a second process that simply aggregates the current results at each node has to be triggered whenever a partial aggregate is computed. The aggregation strategy can be the same or different from the strategy used for computing the final result. Centralized aggregation might be more suitable though due to the reduced interference. The amount of data each node contributes to the result is determined only by the processing speed of the node. Since the work done for partial aggregation is also part of computing the final aggregate, it is important to reuse the result so that the overall execution time is not increased unnecessarily.

3.2 Parallel Sampling

In order to provide any information on the final result, partial aggregates have to be statistically significant. It has to be possible to define and analyze estimators for the final result using partial aggregates. Online aggregation imposes an additional requirement. The accuracy of the estimator has to improve when more data are used in the computation of partial aggregates. In the extreme case of using the entire dataset to compute the partial aggregate, the estimator collapses on the final result. The net effect of these requirements is that the data subset on which the partial aggregate is computed cannot be arbitrarily chosen. Since sampling satisfies these requirements, the standard approach in online aggregation is to choose the subset used for partial aggregation as a random sample from the data.

Centralized sampling. Thus, an important decision that has to be taken when designing an online aggregation system is how to generate random samples. According to the literature [3], there are two methods to generate samples from the data in a centralized setting. The first method is based on using an index that provides the random order in which to access the data. While it does not require any pre-processing, this method is highly inefficient due to the large number of random accesses to the disk. The second method is based on the idea of storing data in random order on disk such that a sequential scan returns random samples at any position. Although this method requires considerable pre-processing at loading time to permute data randomly, it is the preferred randomization method in online aggregation systems [35] since the cost is paid only once and it can be amortized over the execution of multiple queries—the indexing method incurs additional cost for each query. As a result, PF-OLA implements a parallel version of the random shuffling method.

Sampling synopses. It is important to make the distinction between the runtime sampling methods used in online aggregation and estimation based on static samples

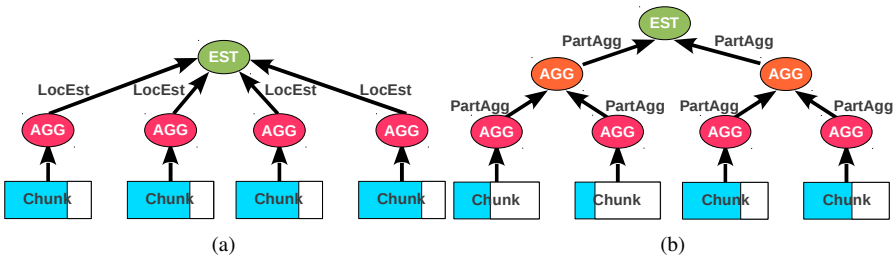


Fig. 1: (a) Centralized aggregation with multiple estimators. (b) Aggregation tree with single estimator.

taken offline [12, 15], i.e., sampling synopses. In the later case, a sample of fixed size is taken only once and all subsequent queries are answered using the sample. This is typically faster than executing sampling at runtime, during query processing. The problem is that there are queries that cannot be answered from the sample accurately enough, for example, highly selective queries. The only solution in this case is to extract a larger sample entirely from scratch which is prohibitively expensive. The sampling methods for online aggregation described previously avoid this problem altogether due to their incremental design that degenerates in a sample consisting of the entire dataset in the worst case.

Sample size. Determining the correct sample size to allow for accurate estimations is an utterly important problem in the case of sampling synopses and iterative online aggregation. If the sample size is not large enough, the entire sampling process has to be repeated, with unacceptable performance consequences. While there are methods that guide the selection of the sample size for a given accuracy in the case of a single query, they require estimating the variance of the query estimator—an even more complicated problem. In the case of overlapped online aggregation, choosing the sample size is not a problem at all since the entire dataset is processed in order to compute the correct result. The only condition that has to be satisfied is that the data seen up to any point during processing represent a sample from the entire dataset. As more data are processed towards computing the query result, the sample size increases automatically. Both runtime sampling methods discussed previously satisfy this property.

Stratified sampling. There are multiple alternatives to obtain a sample from a partitioned dataset—the case in a parallel setting. The straightforward solution is to consider each partition independently and to apply centralized sampling algorithms inside the partition (Figure 1a). This type of sampling is known as *stratified sampling* [9]. While stratified sampling generates a random sample for each partition, it is not guaranteed that when putting all the local samples together the resulting subset is a random sample from the entire data. For this to be the case, it is required that the probability of a tuple to be in the sample is the same across all the partitions. The

immediate solution to this problem is to take local samples that are proportional with the partition size.

Algorithm 1 *RandomSplit* (Dataset D on node n)

Input: Number of nodes N; Random hash function h
Output: Partition $P = \{D_1, D_2, \ldots, D_N\}$ of D

1. **for all** items $d \in D$ **do**
2. Let $k = h(d)$
3. Add d to set D_k in partition P
4. **end for**

Global randomization. A somehow more complicated solution is to make sure that a tuple can reside at any position in any partition—*global randomization* (Figure 1b). This can be achieved by randomly shuffling the data across all the nodes—as a direct extension of the similar centralized approach. The global randomization process consists of two stages, each executed in parallel at every node. In the first stage (Algorithm 1), each node partitions the local data into sets corresponding to all the other nodes in the environment. The assignment of an item to a partition is based on a random hash function which requires as argument a random value independent of the item in order to randomize the data. The index of the item in the dataset (round-robin partitioning) might be a good choice as long as the set assignment does not follow a predictable pattern. An even better choice is a random value generated for the item.

Algorithm 2 *RandomPermutation* (Data fragments from all N nodes at node n)

Input: Set $P^n = \bigcup_{i=1}^{N} D_i^n$ of data fragments from all N nodes; Random number generator *rng*
Output: Random permutation of P^n

1. **for all** items $p \in P^n$ **do**
2. Let $r_p =$*rng*(p) be a random number for item p
3. **end for**
4. Sort P^n in the increasing order of r_p

In the second stage of the randomization process (Algorithm 2), each node generates a random permutation of the data received from all the other nodes—random shuffling. This is required in order to separate the items received from the same origin. The standard method for random shuffling consists in generating a random value for each item in the dataset and then sorting the items according to these random values. Notice that using the random values from the origin node is not guaranteed to produce a random permutation across all the sets.

The main benefit provided by global randomization is that it simplifies the complexity of the sampling process in a highly-parallel asynchronous environment. This in turn allows for compact estimators to be defined and analyzed—a single estimator across the entire dataset. It also supports more efficient sampling algorithms that require a reduced level of synchronization, as is the case with our estimator.Moreover, global randomization has another important characteristic for online aggregation—it

allows for incremental sampling. What this essentially means is that in order to generate a sample of a larger size starting from a given sample is enough to obtain a sample of the remaining size. It is not even required that the two samples are taken from the same partition since random shuffling guarantees that a sample taken from a partition is actually a sample from the entire dataset. Equivalently, to get a sample from a partitioned dataset after random shuffling, it is not necessary to get a sample from each partition.

While random shuffling in a centralized environment is a time-consuming process executed in addition to data loading, global randomization in a parallel setting is a standard hash-based partitioning process executed as part of data loading. Due to the benefits provided for workload balancing and for join processing, hash-based partitioning is heavily used in parallel data processing even without online aggregation. Thus, we argue that global randomization for parallel online aggregation is part of the data loading process and it comes at no cost with respect to sampling.

3.3 Estimation

While designing sampling estimators for online aggregation in a centralized environment is a well-studied problem, it is not so clear how these estimators can be extended to a highly-parallel asynchronous system with data partitioned across nodes. To our knowledge, there are two solutions to this problem proposed in the literature. In the first solution, a sample over the entire dataset is built from local samples taken independently at each partition. An estimator over the constructed sample is then defined. We name this approach *single estimator*. In the single estimator approach, the fundamental question is how to generate a single random sample of the entire dataset from samples extracted at the partition level. The strategy proposed in [34] requires synchronization between all the sampling processes executed at partition level in order to guarantee that the same fraction of the data is sampled at each partition. To implement this strategy, serialized access to a common resource is required for each item processed. This results in a factor of four increase in execution time when estimation is active (see the experimental evaluation section).

In the second solution, which we name *multiple estimators* (Figure 1a), an estimator is defined for each partition. As in stratified sampling theory [9], these estimators are then combined into a single estimator over the entire dataset. The solution proposed in [24] follows this approach. The main problem with the multiple estimators strategy is that the final result computation and the estimation are separated processes with different states that require more complicated implementation.

We propose an asynchronous sampling estimator specifically targeted at parallel online aggregation that combines the advantages of the existing strategies. We define our estimator as in the single estimator solution, but without the requirement for synchronization across the partition-level sampling processes which can be executed independently (Figure 1b). This results in much better execution time. When compared to the multiple estimators approach, our estimator has a much simpler implementation since there is complete overlap between execution and estimation. In this section, we analyze the properties of the estimator and compare it with the two

estimators it inherits from. Then, in Section 5 we provide insights into the actual implementation in PF-OLA, while in Section 6 we present experimental results to evaluate the accuracy of the estimator and the runtime performance of the estimation.

3.3.1 Generic Sampling Estimator

To design estimators for the parallel aggregation problem we first introduce a generic sampling estimator for the centralized case. This is a standard estimator based on sampling without replacement [9] that is adequate for online aggregation since it provides progressively increasing accuracy. We define the estimator for the simplified case of aggregating over a single table and then show how it can be generalized to `GROUP BY` and general SPJ queries (Equation 1) in Section 5.

Consider the dataset D to have a single partition sorted in random order. The number of items in D (size of D) is $|D|$. While sequentially scanning D, any subset $S \subseteq D$ represents a random sample of size $|S|$ taken without replacement from D. We define an estimator for the aggregate as follows:

$$X = \frac{|D|}{|S|} \sum_{s \in S, \mathtt{P}(s)} \mathtt{f}(s) \tag{2}$$

where X has the properties given in Lemma 1:

Lemma 1 *X is an unbiased estimator for the aggregation problem, i.e., $E[X] = \sum_{d \in D, \mathtt{P}(d)} \mathtt{f}(d)$, where $E[X]$ is the expectation of X. The variance of X is equal to:*

$$Var(X) = \frac{|D| - |S|}{(|D| - 1)|S|} \left[|D| \sum_{d \in D, \mathtt{P}(d)} \mathtt{f}^2(d) - \left(\sum_{d \in D, \mathtt{P}(d)} \mathtt{f}(d) \right)^2 \right] \tag{3}$$

It is important to notice the factor $|D| - |S|$ in the variance numerator which makes the variance to decrease while the size of the sample increases. When the sample is the entire dataset, the variance becomes zero, thus the estimator is equal to the exact query result. The standard approach to derive confidence bounds [21,20, 13] is to assume a normal distribution for estimator X with the first two frequency moments given by $E[X]$ and Var (X). The actual bounds are subsequently computed at the required confidence level from the cumulative distribution function (cdf) of the normal distribution. Since the width of the confidence bounds is proportional with the variance, a decrease in the variance makes the confidence bounds to shrink.

A closer look at the variance formula in Equation 3 reveals the dependency on the entire dataset D through the two sums over all the items $d \in D$ that satisfy the selection predicate `P`. Unfortunately, when executing the query we have access only to the sampled data. Thus, we need to compute the variance from the sample. We do this by defining a variance estimator, $\text{Est}_{\text{Var}(X)}$, with the following properties:

Lemma 2 *The estimator*

$$Est_{Var(X)} = \frac{|D|(|D| - |S|)}{|S|^2(|S| - 1)} \left[|S| \sum_{s \in S, \mathtt{P}(s)} \mathtt{f}^2(s) - \left(\sum_{s \in S, \mathtt{P}(s)} \mathtt{f}(s) \right)^2 \right] \quad (4)$$

is an unbiased estimator for the variance in Equation 3.

Having the two estimators X and $\text{Est}_{\text{Var}(X)}$ computed over the sample S, we are in the position to provide the confidence bounds required by online aggregation in a centralized environment. The next step is to extend the generic estimators to the parallel setting of the PF-OLA framework where data is partitioned across multiple processing nodes.

3.3.2 Single Estimator Sampling

In the PF-OLA framework, the dataset D is partitioned across N processing nodes, i.e., $D = D_1 \cup D_2 \cup \cdots \cup D_N$. A sample S_i, $1 \leq i \leq N$, is taken independently at each node. These samples are then put together in a sample $S = S_1 \cup S_2 \cup \cdots \cup S_N$ over the entire dataset D. To guarantee that S is indeed a sample from D, in the case of the synchronized estimator in [34] it is enforced that the sample ratio $\frac{S_i}{D_i}$ is the same across all the nodes. For the estimator we propose, we let the nodes run independently and only during the partial aggregation stage we combine the samples from all the nodes as S. Thus, nodes operate asynchronously at different speed and produce samples with different size. The global randomization guarantees though that the combined sample S is indeed a sample over the entire dataset. As a result, the generic sampling estimator in Equation (2) can be directly applied to this distributed setting without any modifications.

3.3.3 Multiple Estimators Sampling

For the multiple estimators strategy, the aggregate $\sum_{d \in D, \mathtt{P}(d)} \mathtt{f}(d)$ can be decomposed as $\sum_{i=1}^{N} \sum_{d \in D_i, \mathtt{P}(d)} \mathtt{f}(d)$, with each node computing the sum over the local partition in the first stage followed by summing-up the local results to get the overall result in the second stage. An estimator $X_i = \frac{|D_i|}{|S_i|} \sum_{s \in S_i, \mathtt{P}(s)} \mathtt{f}(s)$ is defined for each partition based on the generic sampling estimator in Equation (2). We can then immediately infer that the sum of the estimators X_i, $\sum_{i=1}^{N} X_i$, is an unbiased estimator for the query result and derive the variance $\text{Var}\left(\sum_{i=1}^{N} X_i\right) = \sum_{i=1}^{N} \text{Var}\left(X_i\right)$ if the sampling process across partitions is independent. Since the samples are taken independently from each data partition, local randomization of the data at each processing node is sufficient for the analysis to hold.

3.3.4 Discussion

We propose an estimator for parallel online aggregation based on the *single estimator* approach. The main difference is that our estimator is completely asynchronous and

allows fully parallel evaluation. We show how it can be derived and analyzed starting from a generic sampling estimator for centralized settings. The implementation in PF-OLA for three different aggregation problems of various complexity is presented in Section 5. We conclude with a detailed comparison with a stratified sampling estimator (or *multiple estimators*) along multiple dimensions:

- *Data randomization.* While the multiple estimators approach requires only local randomization, the single estimator approach requires global randomization across all the nodes in the system. Although this might seem a demanding requirement, the randomization process can be entirely overlapped with data loading as part of hash-based data partitioning.
- *Dataset information.* Multiple estimators requires each node to have knowledge of the local partition cardinality, i.e., $|D_i|$. Single estimator needs only full cardinality information, i.e., $|D|$, where the estimation is invoked.
- *Accuracy.* According to the stratified sampling theory, multiple estimators provides better accuracy when the size of the sample at each node is proportional with the local dataset size (but not a requirement) [9]. This is not true in the general case though with the variance of the estimators being entirely determined by the samples at hand. Given that PF-OLA is a highly asynchronous framework, this optimal condition is hard to enforce.
- *Convergence rate.* As with accuracy, it is not possible to characterize the relative convergence rate of the two methods in the general case. Nonetheless, we can argue that multiple estimators is more sensitive to discrepancies in processing across the nodes since the effect on variance is only local. Consider for example the case when one variance is considerably smaller than the others. Its effect on the overall variance is asymptotically limited by the fraction it represents from the overall variance rather than the overall variance.
- *Fault tolerance.* The effect of node failure on the estimation process is catastrophic for multiple estimators. If one node cannot be accessed, it is impossible to compute the estimator and provide bounds since the corresponding variance is infinite. For single estimator, the variance decrease stops at a higher value than zero. This results in bounds that do not collapse on the true result even when all the available data is processed.

4 PF-OLA Framework Design

Online aggregation in PF-OLA is a process consisting of multiple stages. In the first stage, data are stored in random order on disk. This is an offline process executed at load time. Once data are available, we can start executing aggregate queries. A query specifies the input data and the computation to be executed. Every computation, including estimation, is expressed as a UDA—the central abstraction in the PF-OLA framework. The user application submits the query to a coordinator—a designated node managing the execution. It is the job of the coordinator to send the query further to the worker nodes, coordinate the execution, and return the result to the user application once the query is done. The result of a query is always a UDA containing the final state of the aggregate—PF-OLA takes UDAs as input and produces a UDA as

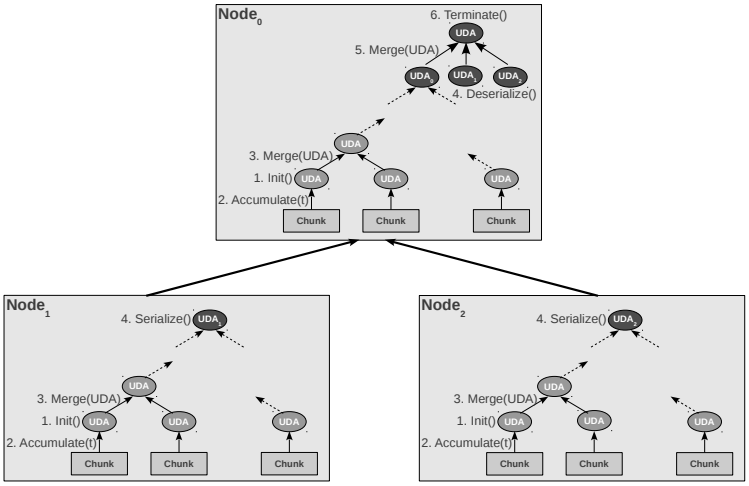


Fig. 2: UDA evaluation mechanism.

output by implementing the UDA evaluation mechanism. When executing the query in non-interactive mode, the user application blocks until the final UDA arrives from the coordinator. When running in interactive mode with online aggregation enabled, the user application emits requests to the coordinator asking for the current state of the computation. Evidently, the returned UDA has an incomplete state since it is computed only over a portion of the data. The user application can use this partial UDA in many different ways. Computing online estimators and confidence bounds is only one of them.

To our knowledge, PF-OLA is the first online aggregation system in which the estimation is driven by the user application rather than the system. The main reason for this design choice is our goal to allow maximum asynchrony between the nodes in the system and to minimize the number of communication messages. A query always starts executing in non-interactive mode. Partial results are extracted asynchronously based only on user application requests. It is clear that generating partial results interferes with the actual computation of the query result. In the case of aggregate computation though, this is equivalent to early aggregation which is executed as part of the final aggregation nonetheless. Given this high-level description of the framework, we concentrate our attention on the following two important questions:

- How to enhance the UDA interface with estimation functionality without modifying the execution model?
- How to optimally overlap the estimation process with query execution?

4.1 User-Defined Aggregates (UDA)

UDAs [32] represent a mechanism to extend the functionality of a database system with application-specific aggregate operators similar in nature to user-defined

data types (UDT) and user-defined functions (UDF) [29]. A UDA is typically implemented as a class with a standard interface defining the following four methods [10]: `Init`, `Accumulate`, `Merge`, and `Terminate`. These methods operate on the `state` of the aggregate which is also part of the class. While the interface is standard, the user has complete freedom when defining the `state` and implementing the methods. The execution engine (runtime) computes the aggregate by scanning the input relation and calling the interface methods as follows. `Init` is called to initialize the state before the actual computation starts. `Accumulate` takes as input a tuple from the input relation and updates the state of the aggregate according to the user-defined code. `Terminate` is called after all the tuples are processed in order to finalize the computation of the aggregate. `Merge` is not part of the original specification [32] and is intended for use when the input relation is partitioned and multiple UDAs are used to compute the aggregate (one for each partition). It takes as parameters two UDAs and it merges their states into the state of an output UDA. In the end, all the UDAs are merged into a single one upon which `Terminate` is called. A graphical depiction of the entire execution process is shown in Figure 2.

In order to provide online estimates, it is clear that additional data need to be stored in the UDA `state`. The exact data are determined by the actual estimator. Since users implement the UDA, they have complete control over what data go in the `state`. What users do not have control over though is the estimation process, driven by the UDA interface and the invocation mechanism. This imposes restrictions on the set of estimators that can be implemented using the standard UDA interface. Thus, the UDA interface also needs to be extended with additional functions. Our goal is to limit the modifications to the UDA interface and to the overall invocation process while still allowing for any estimator to be implemented in the PF-OLA framework.

One of the main contributions made in this paper is the design of extensions to the UDA interface for online aggregation and the subsequent implementation in PF-OLA. The first extension handles the communication problem. Since UDAs are transferred between nodes and from the coordinator to the user application, a mechanism that does this transparently is required. Given that the UDA state is defined by the user, it is the writer of the UDA who is in the position to specify what needs to be transferred in order to re-create an equivalent UDA in the memory space of another process. Thus, it is natural to apply the same principles at the core of UDA and extend the UDA interface with methods to `Serialize/Deserialize` the UDA state. It is the job of the UDA creator to implement these methods correctly and the responsibility of the framework to invoke them transparently.

The second extension is specifically targeted at estimation modeling for online aggregation. To support estimation, the UDA state needs to be enriched with additional data on top of the original aggregate. Although it is desirable to have a perfect overlap between the final result computation and estimation, this is typically not possible. In the few situations when it is possible, no additional changes to the UDA interface are required. For the majority of the cases though, the UDA interface needs to be extended in order to distinguish between the final result and a partial result used for estimation. As we shall see in Section 5, there are at least two methods that need to be added: `EstimatorTerminate` and `EstimatorMerge`. `EstimatorTerminate` computes a local estimator at each node. It is invoked af-

ter merging the local UDAs during the estimation process. `EstimatorMerge` is called to put together in a single UDA the estimators computed at each node by `EstimatorTerminate`. It is invoked with UDAs originating at different nodes. It is important to notice that `EstimatorTerminate` is an intra-node method while `EstimatorMerge` is inter-node. It is possible to further separate the estimation from aggregate computation and have an intra-node `EstimatorMerge` and an inter-node `EstimatorTerminate`. While this adds more flexibility and might be required for particular estimation models, we have not encountered such a situation.

The third extension we add to the UDA interface is the `Estimate` method. It is invoked by the user application on the UDA returned by the framework as a result of an estimation request. The complexity of this method can range from printing the UDA state to complex statistical models. In the case of online aggregation, `Estimate` computes an estimator for the aggregate result and corresponding confidence bounds. As with the other methods, the only restriction on `Estimate` is that it can only access the UDA state.

Method	**Usage**

`Init ()`
`Accumulate (Item` d`)`
`Merge (UDA` $input_1$`, UDA` $input_2$`, UDA` $output$`)`
`Terminate ()`
	Basic interface

`Serialize ()` `Deserialize ()`	Transfer UDA across processes
`EstimatorTerminate ()` `EstimatorMerge (UDA` $input_1$`, UDA` $input_2$`, UDA` $output$`)`	Partial aggregate computation
`Estimate (`$estimator$`, `$lower$`, `$upper$`, `$confidence$`)`	Online estimation

Table 1: Extended UDA interface.

Table 1 summarizes the extended UDA interface we propose for parallel online aggregation. This interface abstracts both the aggregation and estimation processes in a reduced number of methods, releasing the user from the details of the actual execution in a parallel environment which are taken care of transparently by PF-OLA. Thus, the user can focus only on estimation modeling. In the rest of the paper we assume this interface for all the UDA examples we provide. Whenever a method is missing, it is assumed that it does not change the UDA state. In [32] the authors observed that the basic UDA interface could be applied to online aggregation. As we shall see in Section 5, this is the case only for a reduced class of estimators.

4.2 GLADE

GLADE [30, 8] is a parallel processing system optimized for the execution of Generalized Linear Aggregates (GLA). A GLA is an associative-decomposable UDA. Essentially, what this means is that the order in which `Accumulate` and `Merge`

are invoked does not change the final result. This in turn allows for efficient asynchronous computation in a parallel environment along an aggregation tree (Figure 2). The UDAs that do not satisfy the associative-decomposable condition require strict ordering when merged together, typically enforced through synchronization at a single node in the system—all the UDAs are first gathered at the node and then `Merge` is called in the proper order. GLADE can also handle this more general type of UDA at the expense of a considerable drop in performance. Since the online aggregation formulation (Equation 1) we consider in this paper is associative-decomposable, thus a GLA, we choose GLADE as the runtime environment of our PF-OLA framework. While GLADE can execute GLAs specified using the basic UDA interface, it cannot provide the partial aggregate state required for online estimation. In this section, we show how to extend GLADE in order to handle GLAs represented through the extended UDA interface. This is quite a challenging task when both execution efficiency and estimation convergence have to be optimized—the case in overlapped online aggregation.

To understand how online aggregation is implemented in GLADE, we first show how GLA processing is realized. As with any other parallel system, GLADE consists of two types of nodes—one coordinator and multiple worker nodes. All GLA computations are directed to the coordinator who is responsible for setting-up the data flow between nodes – building the aggregation tree – and managing the execution. The coordinator compiles the GLA code based on the query parameters and then ships it together with a job description to the nodes. The worker nodes have both processing resources as well as attached storage. They act as independent entities, each running a GLA-enhanced instance of the DataPath [5] database system. The node loads the compiled GLA code inside DataPath and then executes it on the local data. As a final step, the resulting GLAs are put together along the tree structure using the `Merge` method in the UDA interface. While all this seems standard parallel database processing, adding GLAs as first class citizens inside a relational database such as DataPath is far from trivial.

4.3 DataPath

DataPath [5] is a centralized multi-query processing database for analytics capable of providing single-query performance even when a large number of (different) queries are running in parallel. The most distinguishable features it brings are a push-based execution model and full data sharing at all stages during query processing. It is the speed at which data are read from disk that drives the entire execution. Data are read from disk asynchronously by multiple parallel threads and pushed into the execution engine for processing. DataPath operators process data at chunk granularity – typically a few million tuples – in highly optimized loops that scan the tuples and call operator-specific tasks. Data parallelism is achieved by processing multiple chunks in parallel—inside the same operator and across operators. Pipelining is implemented both at query plan level – chunks are passed between operators – as well as operator level—operators are decomposed into small tasks that are executed concurrently. The DataPath execution engine has at its core two key components—waypoints and

work units. A waypoint manages a particular type of computation, e.g., selection, join, GLA. It is not executing any query processing by itself. It only delegates a particular task to a work unit. A work unit is a thread from a thread pool – there are a fixed number of work units in the system – that can be configured to execute tasks.

Intuitively, we can think of the waypoint as the control unit in a pipelined microprocessor where each work unit corresponds to a stage in the pipeline. Moreover, to support data parallelism, multiple such pipelines are created dynamically during query execution in what resembles a super-scalar microprocessor. For complete generalization, the same principle is applied at the level of each waypoint, resulting in multiple super-scalar units with different structure and functionality. Unlike in hardware where the configuration is static and it cannot be modified, the DataPath configuration is dynamic even for a given processing task. It adapts dynamically to the actual data flow generated from disk, with threads being assigned to the work units that require processing. If no threads are available, the chunk is dropped, all the computation up to that point is lost, and it has to be re-read from disk. As long as the computation is I/O-bound – the case in databases – this should not happen (at all) frequently.

The entire GLA processing is embedded inside the `GLAWaypoint` which consists of a pipeline of two work units – one corresponding to `Accumulate` and one to `Merge` – that invoke the corresponding methods in the UDA interface. The `GLAWaypoint` is not aware of the exact type of GLA it is executing since all it has to do is to relay chunks and tasks to work units that execute the actual work. It needs to store though the state of the GLA it is computing. More precisely, a list of GLA states is stored to allow multiple chunks to be processed in parallel. Then, the computation of a GLA proceeds as follows:

1. When a chunk needs to be processed, the `GLAWaypoint` extracts a GLA state from the list and passes it together with the chunk to a work unit (WorkUnit_1 in Figure 3). The task executed by the work unit is to call `Accumulate` for each tuple in the chunk such that the GLA is updated with all the tuples in the chunk. If no GLA state is passed with the task, a new GLA is created and initialized (`Init`) inside the task, such that a GLA is always sent back to the `GLAWaypoint`.
2. When all the chunks are processed, the list of GLA states has to be merged. Notice that the maximum number of GLA states that can be created is bounded by the number of work units in the system. The merging of two GLAs is done by another task that calls `Merge` on the two states (WorkUnit_2 in Figure 3).
3. In the end, `Terminate` is called on the last state inside another task submitted to a work unit.

4.4 Parallel Online Aggregation in GLADE

At a high level, enhancing GLADE with online aggregation is just a matter of providing support for GLAs expressed using the extended UDA interface in Table 1. Intuitively, this can be done using the same mechanism we currently have in place inside the `GLAWaypoint`. While this is a good starting point, there are multiple aspects that require careful consideration. For instance, the system is expected to

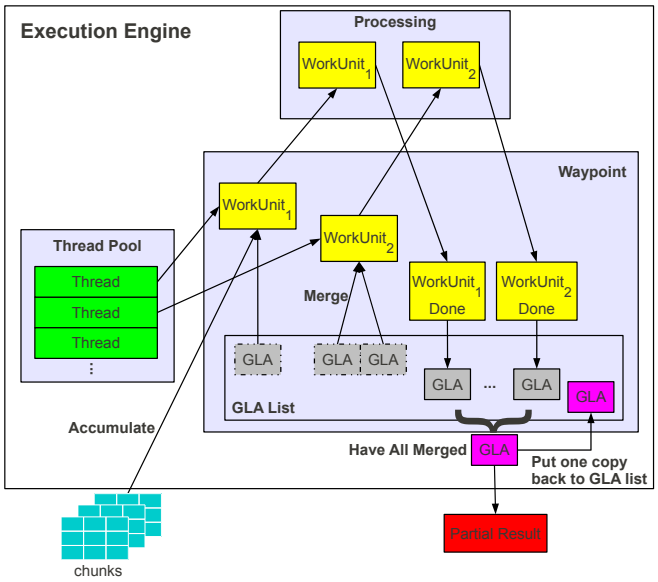


Fig. 3: Execution strategy for parallel online aggregation in PF-OLA.

process partial result requests at any rate, at any point during query execution, and with the least amount of synchronization among the processing nodes. Moreover, the system should not incur any overhead on top of the normal execution when online aggregation is enabled. Under these requirements, the task becomes quite challenging. Our solution overlaps online estimation and actual query processing at all levels of the system and applies multiple optimizations.

We present how online aggregation requests are processed by following the path of the message in the system. In the normal situation, a partial result request follows exactly the same path as a new GLA computation. Problems arise when multiple requests overlap, when requests are sent during the final result aggregation phase, and when there are nodes who have finished local processing and nodes who have not. All these scenarios are likely to appear given the asynchronous nature of the requests and of the processing. Our solution handles each of these situations as follows:

- A partial result request which arrives while a particular node is still processing a previous request is discarded. The partial result generated for the earliest request is returned for all the discarded requests.
- A partial request received at the DataPath execution engine during the local GLA aggregation stage – final result aggregation stage – is discarded given that the same final GLA would be produced nonetheless.
- If a node has finished local processing and a partial result request is received, the node returns the local GLA for merging with the partial/final GLAs from the other nodes. This allows for online estimates to be produced even when there is considerable discrepancy in processing speed across nodes.

Once the request arrives at the `GLAWaypoint`, it is not clear how to proceed. The obvious choice is to process exclusively the partial result computation in order to generate the estimate and the bounds as soon as possible. This results in dropped chunks and thus incurs delays in the final GLA processing. It is though the only available choice in all online aggregation systems we are aware of. The solution we adopt in PF-OLA is to overlap online estimation and GLA processing. Abstractly, this corresponds to executing two simultaneous GLA computations. Given that DataPath is a multi-query processing system optimized for sharing data across multiple queries, optimal performance is expected. To get this performance though, careful consideration of several aspects is required. Rather than treating actual computation and estimation as two separate GLAs, we group everything into a single GLA satisfying the extended UDA interface (Table 1). This simplifies the logic inside the `GLAWaypoint` to the following steps:

- A partial result request triggers the merging of all existent GLAs, those inside the waypoint as well as those modified upon by a work unit. The resulting GLA is the partial result. Unlike the final result which is extracted from the waypoint and passed for further merging across the nodes, a copy of the partial result needs to be kept inside the waypoint and used for the final result computation (Figure 3).
- The newly arriving chunks are processed as before. The result is a completely new list of GLA states. The local GLA resulted through merging is added to this new list once the merging process ends.

To summarize, adding online aggregation to GLADE requires the extraction of a snapshot of the system state that can be used for estimation. Our solution overlaps the process of taking the snapshot with the actual GLA processing in order to have minimum impact on the overall execution time. This is facilitated to some extent by the multi-query processing capabilities of the underlying DataPath system.

5 PF-OLA Estimation Examples

In this section, we show how three different online estimation problems of complexity ranging from simple and `GROUP BY` aggregation to general SPJ queries are expressed in the PF-OLA framework. We present the GLAs for single and multiple estimators, respectively, and discuss the most significant implementation aspects.

5.1 Aggregation

The first online aggregation problem we consider is single-table aggregation. Specifically, we consider aggregates of the following SQL form:

```
SELECT SUM(f(t))
FROM TABLE AS t                    (5)
WHERE P(t)
```

which compute the SUM of function `f` applied to each tuple in table `TABLE` that satisfies condition `P`. It is straightforward to express the aggregate (5) in the GLA

form since it is an associative-decomposable expression. The `state` consists only of the running sum, initialized at zero. `Accumulate` updates the current sum with `f(t)` only for the tuples `t` satisfying the condition `P`, while `Merge` adds the states of the input GLAs and stores the result as the state of the output GLA.

Algorithm 3 *GLASum-SingleEstimator*

State: *sum*; *sumSq*; *count*
Init ()
1. $sum = 0$; $sumSq = 0$; $count = 0$

Accumulate (Tuple `t`)
1. **if** `P(t)` **then**
2. $sum = sum + \texttt{f(t)}$; $sumSq = sumSq + \texttt{f}^2\texttt{(t)}$; $count = count + 1$
3. **end if**

Merge (GLASum $input_1$, GLASum $input_2$, GLASum $output$)
1. $output.sum = input_1.sum + input_2.sum$
2. $output.sumSq = input_1.sumSq + input_2.sumSq$
3. $output.count = input_1.count + input_2.count$

Terminate ()
Estimate (*estimator*, *lowerBound*, *upperBound*, *confLevel*)
1. $estimator = \frac{|D|}{count} * sum$
2. $estVar = \frac{|D|*(|D|-count)}{count^2*(count-1)} * \left(count * sumSq - sum^2\right)$
3. $lowerBound = estimator + NormalCDF\left(\frac{1-confLevel}{2}, \sqrt{estVar}\right)$
4. $upperBound = estimator + NormalCDF\left(confLevel + \frac{1-confLevel}{2}, \sqrt{estVar}\right)$

Algorithm *GLASum-SingleEstimator* implements the estimator we propose. No modifications to the UDA interface are required. Looking at the GLA state, it might appear erroneous that no sample is part of the state when a sample over the entire dataset is required in the estimator definition. Fortunately, the estimator expectation and variance can be derived from the three variables in the state computed locally at each node and then merged together globally. This reduces dramatically the amount of data that needs to be transferred between nodes. To compute the estimate and the bounds, knowledge of the full dataset size is required in `Estimate`.

The GLA implementation for the multiple estimators approach is given in Algorithm *GLASum-MultipleEstimators*. The extended UDA interface has to be used in this case. The standard UDA interface methods are identical to Algorithm *GLASum-SingleEstimator* for single estimator and are not repeated. The `state` consists of variables to compute the query result and local estimators – *sum*, *sumSq*, *count* – and variables to compute the global estimator—*est*, *estVar*. This separation is necessary since the estimation process and the query result computation are not entirely overlapped. The only difference in `Estimate` when compared to the single estimator approach is the scaling factor compensating for the local dataset size rather than the overall size. Method `EstimatorTerminate` is executed at each node to compute the local estimator corresponding to an estimation stage once the local merging finished. The two local estimators *est* and *estVar* are then merged across all the pro-

Algorithm 4 *GLASum-MultipleEstimators*

State: *sum*; *sumSq*; *count*; *est*; *estVar*
EstimatorTerminate ()
1. $est = \frac{|D_i|}{count} * sum$
2. $estVar = \frac{|D_i|*(|D_i|-count)}{count^2*(count-1)} * \left(count * sumSq - sum^2\right)$

EstimatorMerge (GLASum $input_1$, GLASum $input_2$, GLASum $output$)
1. $output.est = input_1.est + input_2.est$
2. $output.estVar = input_1.estVar + input_2.estVar$

Estimate (*estimator*, *lowerBound*, *upperBound*, *confLevel*)
1. $estimator = est$
2. $lowerBound = estimator + NormalCDF\left(\frac{1-confLevel}{2}, \sqrt{estVar}\right)$
3. $upperBound = estimator + NormalCDF\left(confLevel + \frac{1-confLevel}{2}, \sqrt{estVar}\right)$

cessing nodes in `EstimatorMerge` to complete the estimation stage. `Estimate` is then called on the resulting GLA to get the estimator and bounds.

5.2 Group-By Aggregation

The second problem we address is `GROUP BY` aggregation. Specifically, we consider queries of the following SQL form:

```
SELECT gAtts, SUM(f(t))
FROM TABLE AS t
WHERE P(t)
GROUP BY gAtts
```
(6)

which compute the same type of aggregate functions as (5) over the partitions induced by the grouping attributes `gAtts` rather than over the entire table. Notice that query (6) can be rewritten as multiple queries of the form (5), one for each group, with each distinct value of the grouping attributes encoded as a condition in `P`. Based on this formulation, we define an independent estimator for each group and derive confidence bounds accordingly. As long as there is no correlation between the grouping attributes and the aggregate function, the analysis provided for the generic sampling estimator holds. The extension to the PF-OLA parallel setting follows the same principles of single and multiple estimators, respectively, applied to individual groups. The reason we treat `GROUP BY` aggregation separately is to illustrate GLA composition. Instead of having different entities for the `GROUP BY` operator and UDAs, `GROUP BY` is represented as a GLA with composite state containing GLAs for the aggregate function.

Algorithm *GLAGroupBy* gives the implementation of the `GROUP BY` GLA for the single estimator approach. The GLA `state` consists of a hash table that maps the aggregate GLAs to the corresponding grouping attributes. For the purpose of the presentation, the selection predicate `P` is kept inside the aggregate GLA. The estimation logic is encapsulated in the aggregate GLA – in this case *GLASum-SingleEstimator* –

Algorithm 5 *GLAGroupBy*

State: *groups* : *Map(*`gAtts` $\mapsto$ *GLASum-SingleEstimator*)
Init ()
Accumulate (Tuple `t`)
1. **if** `t.gAtts` is not in *groups* **then**
2. Insert `t.gAtts` into *groups*
3. *groups*[`t.gAtts`].$Init()$
4. **end if**
5. *groups*[`t.gAtts`].$Accumulate$(`t`)

Merge (GLAGroupBy $input_1$, GLAGroupBy $input_2$, GLAGroupBy $output$)
1. $output.groups = input_1.groups$
2. **for all** groups $g \in input_2.groups$ **do**
3. **if** g.`gAtts` is in *output.groups* **then**
4. Merge *g.GLA* into *output.groups*[g.`gAtts`]
5. **else**
6. Insert *g* into *output.groups*
7. **end if**
8. **end for**

Terminate ()
Estimate (*estimator*, *lowerBound*, *upperBound*,*confLevel*, *group*)
1. *groups*[*group*].*Estimate*(*estimator*, *lowerBound*, *upperBound*, *confLevel*)

with the *GLAGroupBy* acting as a wrapper that directs the method calls to the correct group (`Accumulate` and `Estimate`). `Merge` invokes merging at the aggregate GLA level whenever a group is present in both input arguments and copies the entries otherwise. Since the GLA for multiple estimators is similar, it is not included.

5.3 Join Group-By Aggregation

Following the trend of increasing complexity, we focus now on the general join `GROUP BY` aggregation problem having the SQL form:

```
SELECT gAtts, SUM(f(t1 • t2))
FROM TABLE1 AS t1, TABLE2 AS t2
WHERE P(t1 • t2)                    (7)
GROUP BY gAtts
```

In order to execute such a query in a parallel database, it is evident that tuples with the same join attributes in the two relations need to reside on the same node. We restrict the query to cases where $\texttt{TABLE}_2$ is a replicated relation across all the nodes, small enough to fit in memory. This allows us to represent the computation as a composite GLA with $\texttt{TABLE}_2$ being part of the GLA `state` together with a *GLAGroupBy* instance. It also simplifies the estimation logic, encapsulated entirely in the *GLAGroupBy*.

Algorithm *GLAJoin* contains the implementation of the aggregate (7) as a GLA based on the traditional UDA interface. In `Init`, the hash table corresponding to $\texttt{TABLE}_2$ is built in memory. Once this blocking process is over, the items in $\texttt{TABLE}_1$

Algorithm 6 *GLAJoin*

State: H : *join hash table*, $groups$: *GLAGroupBy*
Init ()
1. Initialize join hash table H with tuples in $\texttt{TABLE}_2$

Accumulate (Tuple $\texttt{t}_1$)
1. **for all** join tuples $\texttt{t} = \texttt{t}_1 \bullet \texttt{t}_2$ generated by $\texttt{t}_1$ in H **do**
2. $\quad groups.Accumulate(\texttt{t})$
3. **end for**

Merge (GLAJoin $input_1$, GLAJoin $input_2$, GLAJoin $output$)
1. $Merge(input_1.groups, input_2.groups, output.groups)$

Terminate ()
Estimate (*estimator*, *lowerBound*, *upperBound*, *confLevel*, *group*)
1. *groups.Estimate*(*estimator*, *lowerBound*, *upperBound*, *confLevel*, *group*)

are processed by `Accumulate`. First, the join tuples are found in the hash table H and each of them is passed into *GLAGroupBy* for further accumulation. `Merge` and `Estimate` are wrapper calls to corresponding methods in *GLAGroupBy*. Given that hash table H is built during the initialization, it is clear that no estimates can be computed until this phase is over. This is perfectly acceptable if we consider initialization to be part of query setup rather than query execution. Since in the PF-OLA framework H is built by the user application and passed for execution to the processing nodes together with the query, this is the case.

In the following, we discuss generalizations to cases when $\texttt{TABLE}_2$ is not replicated across all the processing nodes and when it is too large to fit in memory. The only solution for parallel join processing is to bring tuples having the same join key on the same node, i.e., partition the tables on the join key. This can be done offline, during data loading, or online, during query execution. Since offline partitioning is the standard procedure, we focus on this case. We argue that as long as the partitioning is based on a random hash function and all the partitions corresponding to one of the two relations fit in memory – assume $\texttt{TABLE}_2$ for consistency – the solution for the replicated case is immediately applicable. The only difference is that each local estimator corresponds to a partition rather than the entire domain. When not all partitions of any of the two relations fit in memory, extensions to the standard ripple join [17] such as SMS join [21] or PR join [7] have to be combined with data partitioning to design adequate estimators. The algorithm proposed in [24] represents such an example. We do not provide details on such extensions since they are beyond the scope of this paper. Nonetheless, we plan to address this topic in future work.

6 Empirical Evaluation

Our experimental study evaluates the main characteristics of the PF-OLA framework:

- The expressiveness to represent different estimation models using the extended UDA interface given in Table 1. For this, we implement GLA instances corresponding to the three estimation models – multiple estimators and single estimator with and without synchronization – for each of the three problems discussed in Section 5 and check the ability of the framework to execute them.

– The lack of noteworthy overhead incurred by the estimation process on top of the actual execution when the estimators are implemented and executed in PF-OLA. For this, we measure the execution time with and without estimation across multiple instances of the three aggregation problems with various degrees of complexity and across different system configurations.

An equally important objective of the study is to provide a thorough comparison between the non-synchronized single estimator solution we propose in this paper and multiple estimators. To this end, we evaluate the "time 'til utility" (TTU) [13] or convergence rate of the estimators as a function of query selectivity, number of groups, and data skew; the correctness of the confidence bounds they produce; the overhead in execution time they incur; and their robustness in the face of worker node delays and failures. The effect of parallelism on the two estimators is also studied by measuring the scaleup across cluster instances with different number of nodes.

Although we also executed the same set of experiments for the synchronized version of the single estimator, we do not include the results in this comparison. The reason is that this estimation model incurs a factor of four delay in execution time and thus has a considerable higher TTU.

6.1 Setup

Implementation. We implemented the PF-OLA framework as part of the GLADE parallel processing system [8]. Since GLADE executes GLA computations specified using the basic UDA interface, we made considerable changes in order to enhance GLADE with support for the extended UDA interface. These changes involved modifications to the core execution engine, mainly the `GLAWaypoint`, as well as adding support for new types of messages at the communication infrastructure level. Overall, somewhere around $10,000$ lines of `C++` code. We emphasize that this is a one time effort to enhance the GLADE framework with online aggregation. Subsequently, the user has to implement only the extended GLA interface in order to design any estimation model—no changes to the system internals are required anymore, as is the case with all the previous online aggregation systems we are aware of. To prove our point, we successfully implemented all the proposed estimation models as GLAs. The result is a system that executes any GLA computation with or without online aggregation, depending on the runtime behavior of the user application.

All the GLAs presented in this study are implemented as standard `C++` classes providing the extended UDA interface and having the state represented as class member variables. The `template` mechanism and class composition are used extensively to avoid re-writing boiler-plate code for every single GLA. The actual computation is specified as a script containing the physical execution plan—input data sources and waypoints, the data flow between them, the association between GLAs and GLA waypoints, and the GLA arguments. Scripting provides a certain level of abstraction specific to a higher-level language and allows for significant code reuse.

System. We executed all our experiments on a 9-node shared nothing cluster. Each node has 2 AMD Opteron 8-core processors for a total of 16 cores running at $2GHz$,

16*GB* of memory, 4 1*TB* hard-drives, and runs Ubuntu 10.10 64-bit. Out of the 16*GB* memory, 12.5*GB* are reserved as $6,000$ 2*MB* huge pages for reduced TLB misses. The disks perform sequential reads at 130*MB/s* according to `hdparm`, for 520*MB/s* total I/O bandwidth at a node. The nodes are mounted inside the same rack and are inter-connected through a Gigabit Ethernet switch. One node is configured as the coordinator and the other 8 nodes are workers.

Data. The dataset used in our experiments is TPC-H [2] scale **8,000** (**8*TB***). Instead of generating the entire 8*TB* at once (impossible due to the 4*TB* limit on each node), we generated 8 1*TB* independent TPC-H scale 1 instances, one on every worker node. Notice that the instances are not copies of the same dataset and they are not treated as replicas—they are a single dataset partitioned into 8 non-overlapping subsets. The data is then stripped across the 4 available disks, for approximately 250*GB* on each disk. Rather than executing a global data randomization process, we opted for local randomization in order to measure scalability accurately. This is supposed to have a negative effect on the accuracy and correctness of the estimator proposed in the paper since it relies on global randomization. Nonetheless, the experimental results – the Monte Carlo simulations in particular – prove this is not the case and show that local randomization is sufficient in the majority of the cases.

Methodology. All our figures depict relative confidence bounds width as a function of time for cluster configurations of 1, 2, 4, and 8 nodes, respectively. Relative confidence bounds width ($\frac{high-low}{estimate}$) is measured as the ratio between the difference of the confidence interval extremes and the estimate. The bounds are computed at 95% confidence level and are depicted as percentage from the estimate. The different cluster configurations test the scaleup of the PF-OLA framework and the effect of parallelism on the estimators. A system scales-up optimally if the execution time remains constant when both the data and the processing capacity increase proportionally, i.e., all curves collapse to zero at the same time in our figures.

6.2 Aggregation

The aggregation tasks we consider are two different instantiations of query Q6 from the TPC-H benchmark:

```
SELECT SUM(l_extendedprice*l_discount)
FROM lineitem
WHERE l_shipdate between [date1,date2] AND
l_discount between [0.02,0.03] AND l_quantity = 1
```

In the first instance, `l_shipdate` takes values in the one year interval beginning on `'1993-02-26'`. This is considered a low selectivity query since a large number of tuples satisfy the selection predicate (approximately 13.3×10^6 out of 48×10^9 for the 8 node configuration). The second instance restricts `l_shipdate` to a single day, `'1993-02-26'`. It is a high selectivity query (or needle in the haystack) since only $35,000$ tuples are part of the result.

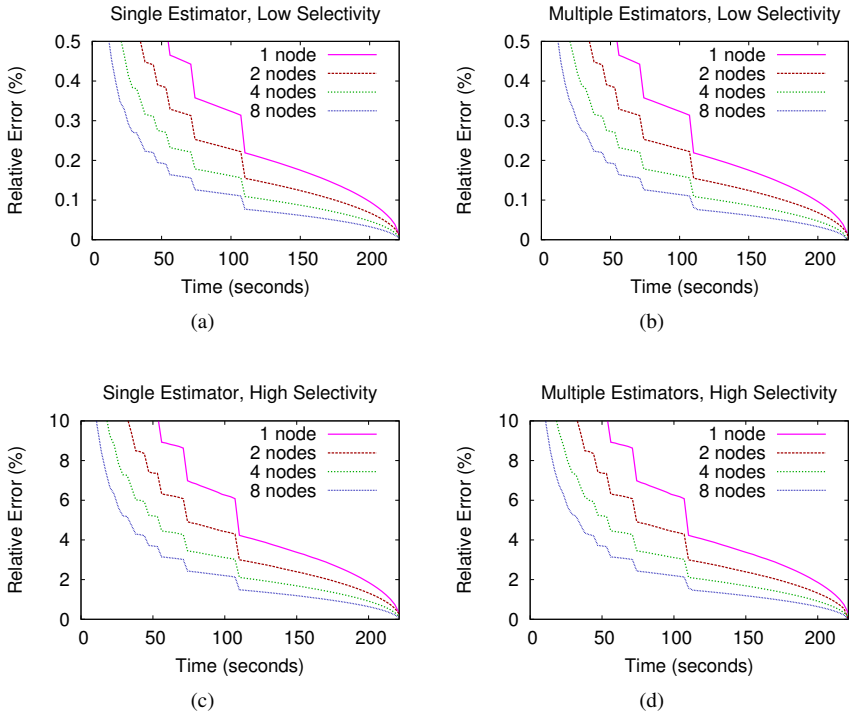


Fig. 4: Comparison between single estimator and multiple estimators for the aggregation task (TPC-H Q6). Figures (a) and (b) correspond to low selectivity (a large number of tuples satisfy the selection predicate), while Figures (c) and (d) correspond to high selectivity (only few tuples satisfy the selection condition).

Figure 4 depicts the relative confidence bounds width as a function of query execution time for the two estimators considered in the study. There are multiple common trends across all the graphs. The width of the bounds decreases as the execution progresses and more tuples are processed, converging to the true result in the end. This is the standard online aggregation behavior. At the same instance in time, the width of the bounds is wider for the configurations with fewer nodes or, equivalently, it takes more time to arrive at the same relative width. This exemplifies perfectly the effect of parallelism on estimation. Having more nodes increases the rate at which result tuples are discovered thus reducing TTU. The execution time for all the configurations is the same. This confirms the scaleup of our framework. The confidence bounds are identical for the two estimators in the single node configuration. This is to be expected given that the two estimators are equivalent in this case. We shall see these trends re-occurring in all the other experiments.

Careful readers might ask why are there steps in the figure? Why the estimator does not converge smoothly to the exact query result? The reason for this behavior

is that a different number of chunks are processed for different estimation intervals. This results in considerable variation in the number of result tuples discovered. A larger number of tuples provide a better variance estimator, thus a steeper decrease in the error. The effect of this phenomenon can be observed better for a smaller number of processing nodes since the variation in the number of chunks is relatively higher to the overall number of chunks. As the processing progresses, the variance estimator stabilizes and the step effect disappears. A possible solution to avoid this behavior is to enforce the same number of chunks at every estimation stage. This requires synchronization though and it deviates from our principle of asynchronous processing.

When comparing the results for the two queries, we observe that the confidence bounds are much tighter in the low selectivity case. Essentially, a relative error of 0.5% is obtained almost from the beginning of the execution in the 8 node configuration. The high selectivity query arrives to the same error only at the end of execution. The explanation is the 3 orders of magnitude difference in the number of result tuples. Nonetheless, an acceptable error of 5% is obtained after less than 50 seconds (20%) in the execution for the same 8 node configuration. This is remarkable if we consider that only a fraction of 0.73×10^{-6} of the overall tuples is part of the result. Although it seems more appropriate to process such a selective query using an index, the complex selection predicate makes building the correct index quite challenging. Thus, we argue that scanning the entire relation and providing estimates as in online aggregation is a more effective approach. Notice also that the execution time in all the figures is the same since the same amount of data is read from disks. In terms of estimators, there is no discernible difference between the single estimator and the multiple estimators solutions. This confirms the good accuracy of the single estimator we propose even when the data is not globally randomized.

6.3 Group-By Aggregation

We consider two group-by aggregation tasks as well. They are both instantiations of query Q1 from TPC-H:

```
SELECT gAtts, SUM(l_quantity),
 SUM(l_extendedprice),
 SUM(l_extendedprice*(1-l_discount)),
 SUM(l_extendedprice*(1-l_discount)*(1+l_tax))
FROM lineitem WHERE l_shipdate between
['1998-09-01','1998-12-01'] GROUP BY gAtts
```

The difference between the two queries is the grouping attributes. In the first instance we use the attributes in Q1, `l_returnflag` and `l_linestatus`. Since this generates only 4 groups, we name this query *group-by small*. The second instance groups by `l_suppkey` for a total of 1 million distinct groups, thus the name *group-by large*.

Figure 5 depicts the results for both estimators considered. Notice that estimates and bounds are computed for all group-aggregate combinations simultaneously. Only one combination is displayed. The relative confidence bounds width for group-by small follows the same pattern as for the low selectivity aggregate query. The interval

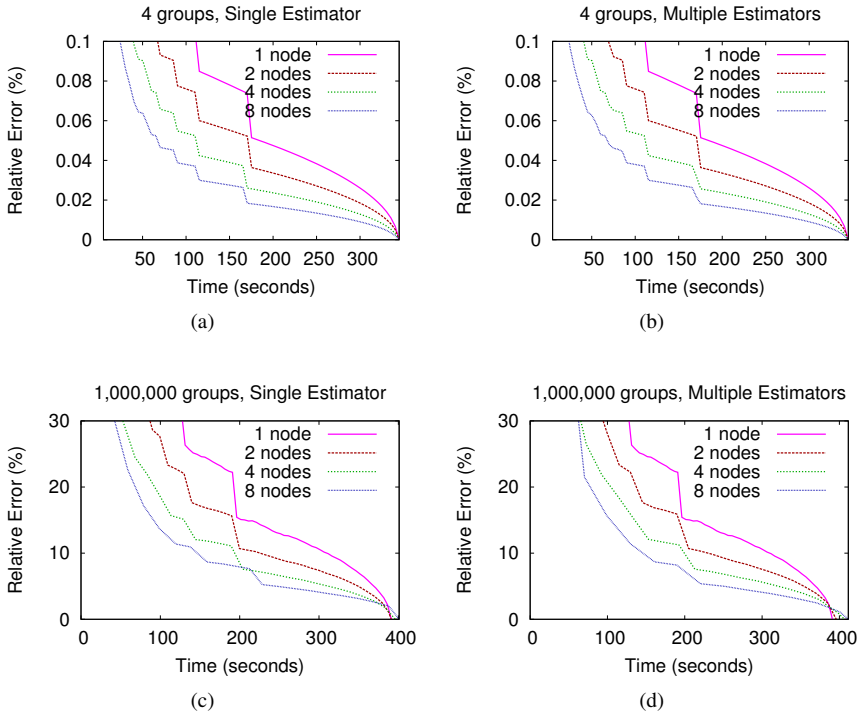


Fig. 5: Comparison between single estimator and multiple estimators for the group-by task (TPC-H Q1). The figures depict relative confidence bounds width as a function of time on cluster configurations of 1, 2, 4, and 8 nodes, respectively. Figures (a) and (b) correspond to the query that generates a small number of distinct groups (4 to be precise), while in Figures (c) and (d) there are 1 million distinct groups. In (a) and (b) we depict the results corresponding to the aggregate `SUM(l_extendedprice*(1-l_discount))` for the `'NF'` group, while in (c) and (d) the aggregate `SUM(l_extendedprice*(1-l_discount)*(1+l_tax))` is shown for `l_suppkey = '1,000,000'`, respectively. The results for the other group-aggregate combinations follow the same pattern.

width is even lower since the number of result tuples is higher. The number of tuples in the result of the group-by large query is very low (3,464 for the displayed group). Finding them while scanning through the entire data takes some time, thus the longer TTU. Still, a relative error of 10% is obtained while only one third in the execution for the 8 node configuration. This is remarkable given the low number of result tuples.

The increase in execution time for the group-by large query is due to the massive state of the GLA. The group-by hash table requires 1 million distinct entries in this case, thus increasing the execution time of all operations in the UDA interface. The

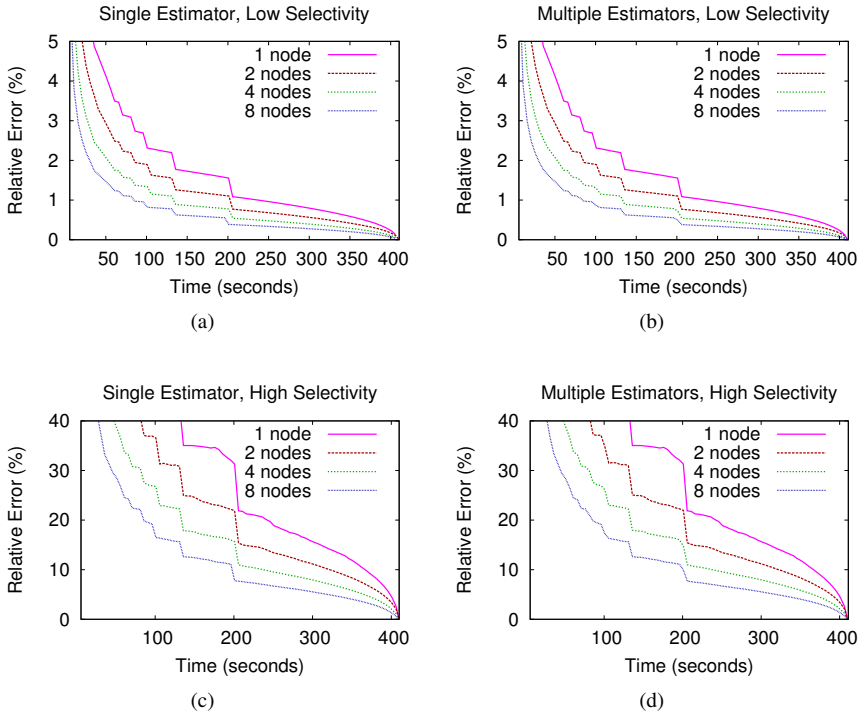


Fig. 6: Comparison between single estimator and multiple estimators for the join group-by task. The figures depict relative confidence bounds width as a function of time on cluster configurations of 1, 2, 4, and 8 nodes, respectively. Figures (a) and (b) correspond to the low selectivity condition, while (c) and (d) correspond to high selectivity as defined in Figure 4. The plots print the results corresponding to `SUM(l_extendedprice*(1-l_discount)*(1+l_tax))` for the `'PERU'` group. The results for the other group-aggregate combinations follow the same pattern.

operation that is most affected though is `Merge`. The shorter execution time for the single node configuration certifies this. As we shall see in the runtime analysis section (Section 6.8) though, the overhead incurred by the estimation process is less than 1% even in this extreme case.

6.4 Join Group-By Aggregation

The join query we use combines the aggregate and group-by queries. It uses the selectivity conditions in the aggregate query and computes the four aggregates in the

group-by query. The exact form is as follows:

```
SELECT n_name, SUM(l_quantity),
 SUM(l_extendedprice),
 SUM(l_extendedprice*(1-l_discount)),
 SUM(l_extendedprice*(1-l_discount)*(1+l_tax))
FROM lineitem, supplier, nation
WHERE l_shipdate between [date1,date2] AND
 l_discount between [0.02,0.03] AND
 l_quantity = 1 AND l_suppkey = s_suppkey AND
 s_nationkey = n_nationkey GROUP BY n_name
```

To execute the query in parallel, `supplier` and `nation` are replicated across all the nodes. They are loaded in memory, pre-joined, and hashed on `s_suppkey`. `lineitem` is scanned sequentially and the matching tuple is found and inserted in the group-by hash table. Merging the GLA states proceeds then as in the group-by case. This join strategy is common in parallel databases.

The results are depicted in Figure 6. A similar trend as in Figure 4 can be observed. This is expected given the similar selectivity conditions. The minor differences are due to the number of tuples in the result. What is remarkable though is the reduced TTU even for extremely selective queries with only $1,000$ tuples in the result (Figure 6c for example). This is a very extreme case where there are no result tuples in many chunks and we still achieve lower than 10% relative error before half of the computation is executed. As with the other queries, there is no discernible difference between single estimator and multiple estimators.

6.5 Data Skew

It is a well-known fact that the TPC-H dataset exhibits a distribution close to uniform. Someone might argue this is the reason why the accuracy of the estimators in the previous experiments is so high. While this argument holds for the low selectivity queries, it is not true for the queries with extremely low selectivity.

To further verify this, we run a series of experiments in a tightly controlled setting meant to study the accuracy as a function of the data skew. For this, we generate a dataset consisting of 3.2×10^9 items distributed over a domain with 10^6 elements using a zipfian distribution with different parameters. The assignment of frequency to domain elements is random. Global randomization is applied to distribute data across the 8 nodes.

Figure 7a and 7b depict the estimator accuracy as a function of the zipfian parameter for the aggregation task. Although we might expect some kind of sensitivity to the skew in the data, the estimators seem to not be affected by the skew. At first sight, this seems unreasonable. After a careful analysis of the experimental conditions, we found the explanation. The first estimator is already computed over 6% of the data. This represents approximately 200×10^6 tuples—a massive sample. At such a sample size, no matter how large and skewed are the data, the accuracy is going to be high. What makes PF-OLA special is that extracting such a large sample takes only

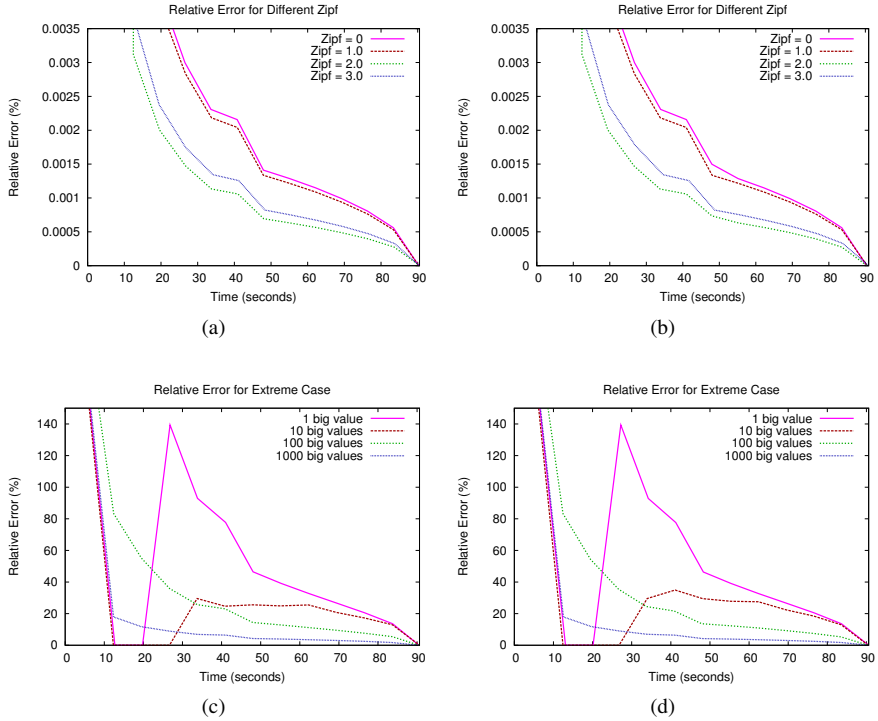


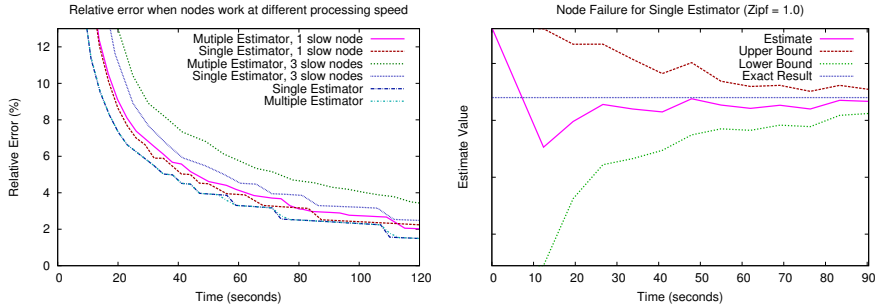
Fig. 7: Effect of data skew on the accuracy of the estimators. (a) and (b) depict the accuracy for datasets generated with different Zipf distributions. (c) and (d) correspond to extreme-case datasets where a few values are orders of magnitude larger than all the other values.

a few seconds. It is practically impossible to get small samples due to the push-based processing implemented at the lowest level of the system.

In the experiments depicted in Figure 7c and 7c, we test the accuracy of the estimators for pathological data distributions. Out of the 3.2×10^9 tuples, all take value 1, except for a handful of outliers which take value 10^9. Sampling accuracy in such situations is known to be extremely poor. The figures confirm this to be the case but only for the most extreme situations where only 10 and, respectively, 1 values are outliers. We do a simple computation to confirm these results. For 100 outliers, the first sample already contains 6 large values with high probability—the exact positions depend on the global randomization. This is more than enough for high accuracy.

6.6 Robustness

We validate the robustness of the estimators presented in the paper in two different scenarios. In the first scenario, one or multiple nodes in the cluster process data at



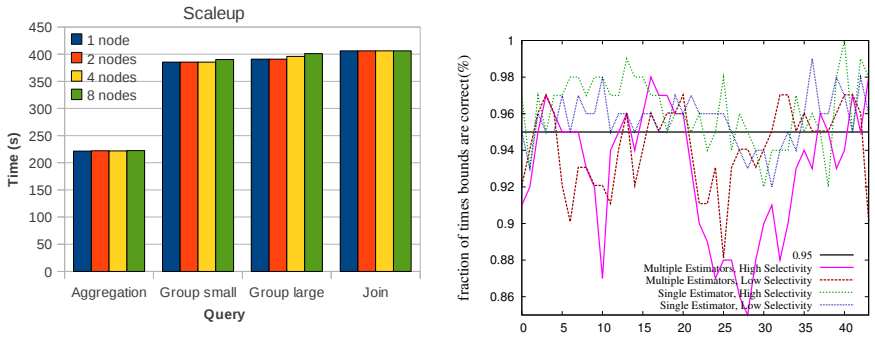
(a) (b)

Fig. 8: (a) Effect of straggler node(s) on estimation. (b) Effect of ”dead” node(s) on the single estimator with global randomization.

a considerable slower rate than the other nodes—they are stragglers. In the second scenario, one node ”dies” and the corresponding data are not available anymore—data are not replicated across nodes.

Figure 8a depicts the relative errors in the scenario where one or three nodes (out of 8) work at slower speed than others and compares the results with the normal configuration when all the nodes work at roughly the same speed. While relative errors are almost the same for the two estimators in the normal case, the curves start to separate when we introduce delays at some of the nodes. We delay a node by intentionally pausing several seconds for every chunk processed. The delays are different across nodes. As a result, the sample size of the delayed node(s) is considerably smaller. This results in the decrease of the accuracy for both estimators. However, single estimator has better accuracy than multiple estimators even though the overall sample size is the same. What is different though is the distribution of the samples across nodes. In the single estimator case, the overall sample increases at a lower rate, thus the accuracy is worse than in the normal configuration. For multiple estimators, different sample sizes at nodes affect only the local estimator. While this might not be considerable for a single node taken separately, the effect is amplified when all the nodes are considered together. Concretely, the variance of the slow nodes is larger. When adding all the variances together to obtain the overall variance for the multiple estimators, we end up with a larger value than the variance of the single estimator computed from a smaller sample size.

In the second scenario, we consider only seven nodes in the estimation process. We still want to estimate the result over all the data, including the missing partition. For multiple estimators, this is simply not possible because we cannot get any estimate on the missing data. In the single estimator case, samples are always drawn from the overall data, independent of the number of available partitions. In this particular case, the maximum sample size is limited to the available data. As a result, the confidence bounds do not collapse on the true result anymore even at the end of query execution (Figure 8b).



(a) (b)

Fig. 9: (a) Query execution scaleup. (b) Monte Carlo simulations to check confidence bounds correctness.

6.7 Correctness

To verify the correctness of the confidence bounds produced by the two estimators, we execute 100 Monte Carlo trials with different data randomizations for the aggregation queries. The 8 node configuration is used. Figure 9b plots the number of times the correct result is between the estimated confidence bounds at 5 second intervals during query execution (there are 43 such points over 222 seconds). The confidence level is set at 95%, thus the additional horizontal line. From the figure, we observe that only the bounds for the single estimator approach verify the required confidence level (they are centered around the 0.95 horizontal line). Multiple estimators has a considerable drop during the second part of the query execution. The reason for this is the asynchrony in execution that starts to affect the estimator. Though we use the same configuration across all the nodes, the minor difference in the number of samples that have been processed at each node starts to accumulate and finally leads to an increasing difference in estimates among nodes. Thus, the multiple estimators method is more prone to generate errors when we combine the estimates together according to the stratified sampling theory [9] since stratified sampling is not in the optimal regime—the samples at nodes are not proportional to the size of the strata. This behavior also proves that the single estimator we propose is more stable and a better solution for parallel online aggregation. We further confirm this with experiments that test the robustness of the estimators.

6.8 Speed & Scalability

Table 2 contains the execution time for the 8 node configuration. It confirms that online aggregation can be enabled for any query at virtually no overhead. As far as we know, PF-OLA is the first online aggregation system achieving this goal. It is the asynchronous overlapping of execution and estimation in the GLADE parallel multi-query processing system that makes this possible. If the synchronized single estimator

proposed in [34] is used, the execution time for the aggregation task is 883 seconds, a factor of 4 increase on top of the normal execution. Clearly, this is not acceptable since our goal is to not increase the execution time beyond normal execution. For this reason, we do not even include the results for the other tasks in Table 2.

Query	Execution Time (seconds)		
	No estimation	Single estimator	Multiple estimators
Aggregate	222	222	222
$\text{Group}_{\text{small}}$	344	345	344
$\text{Group}_{\text{large}}$	404	407	407
Join	409	411	411

Table 2: Execution time.

To clarify the surprisingly low execution time, we analyze in detail the time it takes to execute the group-by small query. Remember that we have TPC-H scale $1,000$ data on each node. lineitem has 6×10^9 tuples for this instance. This corresponds to 1.5×10^9 tuples per disk. DataPath uses columnar storage, thus it reads only the columns required by the query. In this case it reads 7 columns, summing-up to 28 bytes per tuple and 42*GB* per disk. Given the 130*MB/s* disk bandwidth, the theoretical execution time is 323 seconds. This comes close to the actual execution time of 344 seconds, but is not exactly the same. Here are the reasons. We monitored the disk bandwidth during the execution and it was only 123*MB/s*. According to iostat the disk was fully utilized, thus the execution was I/O-bound. With this bandwidth, the execution time is 341.5 seconds. We already knew that the difference is taken by the time to setup the query, thus everything makes perfect sense. It is possible to run similar analyses for the other queries. More parameters such as the time to merge two GLAs, the time to serialize/deserialize the GLA state, and the time to transfer the data between nodes might need to be accounted for though.

When the execution is not I/O-bound, e.g., in-memory databases or SSD storage, the interaction between normal computation and estimation is more complicated since the same processing resources have to be split between these two tasks. We envision a scheduling policy which allows the user to specify the priority of each task. This policy can be customized to control how many computational resources are allocated for estimation and how many for normal execution. Since the two processes overlap in terms of data access, as long as a sufficient level of parallelism is available, the effect on the execution time is minimal.

Although the scaleup of PF-OLA can be inferred from the accuracy figures – the execution time corresponding to all the configurations is the same – Figure 9a shows explicitly the time it takes to run each task on 1, 2, 4, and 8 nodes when the size of the data increases with a corresponding factor. Since the execution time remains constant, this confirms that PF-OLA scales-up linearly. An interesting question is what is the effect of estimation on scalability on larger clusters? Will the estimation overhead become an important fraction from the overall execution time such that

linear scale-up is not achievable anymore? We argue this is not the case since the same communication overhead estimation incurs, normal execution will also incur. Moreover, since normal execution has higher priority, the estimation overhead does not affect the overall execution time.

6.9 Discussion

The main findings of the experimental study are:

- The study proves the expressiveness of the PF-OLA framework. Two estimation models are created for three different tasks, each consisting of two queries. They are all implemented as GLAs with the extended UDA interface and executed successfully by the framework.
- The proposed single estimator has similar TTU and identical or better accuracy than multiple estimators. PF-OLA is able to provide accurate estimations for the query result early in the execution even in the most difficult scenarios when only a handful of tuples are part of the result or when data are skewed.
- The asynchronous single estimator we propose is highly insensitive to processing node delays and failure. This is the signature characteristic when compared to the multiple estimators solution.
- No significant overhead is incurred by online aggregation for any of the queries. The execution is always I/O-bound.
- PF-OLA has perfectly linear scaleup as the data and the processing resources increase proportionally.
- The correctness of the estimators is confirmed through Monte Carlo simulations.

7 Related Work

There is a plethora of work on online aggregation published in the database literature [12] starting with the seminal paper by Hellerstein et al. [18]. We can broadly categorize this work into system design [6,20,31,13], online join algorithms [17,21, 7], online algorithms for estimations other than join [22,36,33], and methods to derive confidence bounds [16]. All of this work is targeted at single-node centralized environments. The parallel online aggregation literature is not as rich though. We identified only a relatively small number of research papers that are closely related to our work. We discuss them in details in the following.

Luo et al. [24] extend the centralized ripple join algorithms [17] to a parallel setting. The proposed parallel hash ripple join algorithm is a non-blocking version of the parallel hybrid hash join algorithm that allows for estimates to the final query result to be computed. This solution requires synchronized processing since the estimation is entirely driven by the tuples found on the normal execution path. The two processes are not separated as is the case in PF-OLA. To devise an asynchronous version of this algorithm, two separate execution paths have to be defined—one for the normal processing and one for estimation. Data traverse the paths independently and asynchronously. We are currently exploring such a parallel hash ripple join strategy in PF-OLA. A stratified sampling estimator [9] is defined to compute the result

estimate while confidence bounds cannot always be derived. We implement a similar stratified sampling estimator in PF-OLA and compare it with the estimator we propose. Our focus is on analyzing the properties of the two estimators along a larger set of dimensions, including robustness, which is not discussed at all in [24]. Moreover, the prototype system implementing the specific parallel hash ripple join algorithm is very particular to the proposed estimator. There is no common framework proposed for general parallel online aggregation.

Wu et al. [34] extend online aggregation to distributed point-to-point (P2P) networks. They introduce a synchronized sampling estimator over partitioned data that requires data movement from storage nodes to processing nodes. We also implement this estimator in PF-OLA and show the poor performance it achieves in a highly-parallel asynchronous system. Since this method is an iterative online aggregation strategy that synchronizes the execution at the end of each iteration, it is not possible to devise an asynchronous version for this solution. In subsequent work [35], Wu et al. tackle online aggregation over multiple queries. They emphasize the benefits of global randomization as a sample generating method.

Recently, online aggregation in Map-Reduce emerged as a popular research area. This is mostly motivated by the poor performance of the Map-Reduce implementation in Hadoop [1], which makes online aggregation a necessity rather than a luxury in the context of Big Data. Hadoop Online (HOP) [11] is an extension to the Hadoop framework which allows for partial aggregates to be extracted during execution using pipelining between operators. Stock Hadoop contains only blocking operators that materialize the intermediate results for fault-tolerance. As explained in Section 3.1, partial result extraction is only the basic requirement for online aggregation. Sampling and estimation provide significance to the partial results. HOP is limited only to partial result extraction. There is no formal sampling or estimation involved. Thus, HOP is not an online aggregation system. It is a partial aggregation system.

It is the work in [25] where HOP is elevated to a real online aggregation system by providing an estimation mechanism. The proposed solution is a Bayesian framework to handle the correlation between the time to process a data partition and the result it generates. This is required because chunks are treated as black boxes that only produce an aggregate. There is no information on what operation was performed to generate the aggregate or on the content of the chunk. Based on the aggregates produced by the processed chunks and the time it took to schedule and process the chunk, a prediction is made for the aggregates in the chunks not scheduled yet—the processed chunks are an independent and identically distributed (iid) sample from the entire chunk population. The Bayesian model is continuously updated as more chunks are processed. This results in more accurate estimates as more data are processed. Although this estimation model is not based on sampling, it can still be expressed as a GLA using the extended UDA interface. We plan to do this in future work to further validate the expressiveness of the PF-OLA framework.

BlinkDB [4] stores pre-computed samples of different sizes on disk. This requires a significant amount of additional storage on top of the original data which might be a problem if the dataset is massive. Moreover, the time to compute the samples – even if executed offline – might be prohibitive. Similar to iterative online aggregation, a query is evaluated on a chosen sample and an estimate is produced. If the accuracy is

not satisfying, a subsequent query can be executed on a larger sample—in the worst case, the query is executed on the entire dataset. While this allows for discrete estimates of increasing accuracy, it cannot support continuous estimation. Determining the correct sample to execute the query on is a complicated problem that requires estimating the variance of the result. None of these problems arise in PF-OLA as long as global randomization is executed on the data. This is a considerably less time-consuming process than taking samples of progressively increasing sizes. A somehow similar idea is used in EARL [23] where a single sample is pre-computed. Bootstrapping is then used to extract multiple samples from the pre-computed sample and compute estimates. The sample sizes are increased dynamically to provide better accuracy. Different from BlinkDB, if the large sample in EARL has to be re-computed, this is done dynamically at runtime.

Different estimation algorithms are proposed in each reference we mention. No common framework for estimation exists. PF-OLA provides a common framework to model a much larger class of estimation models. In terms of performance, all the estimation methods incur considerable overhead. The only exception is PR-Join [7] which combines a non-blocking join algorithm with temporary storage on solid-state drives (SSD) to produce the result tuples much faster, thus increasing the convergence rate. It is a centralized algorithm though.

The last pieces of work we mention are our publications on parallel online aggregation and sampling estimators for parallel online aggregation. [26] is an electronic technical report that contains an evolving description of the PF-OLA framework. The report is continuously updated as more features are added to PF-OLA. This article is based on the January 2013 version of the report. [27] contains a detailed description of sampling estimators for parallel online aggregation. The preliminaries and the estimator analysis (Section 2 and 3) are inspired by the material in [27]. Finally, [28] presents PF-OLA real use-cases in a demonstration scenario.

8 Conclusions

We present PF-OLA, the first framework for parallel online aggregation that does not incur any noticeable overhead on top of the actual computation. The extensive use of parallelism at all levels of the system and the sound overlapping between computation and estimation make this possible. PF-OLA provides an abstract interface enhancing the well-known UDA to express any estimation model. The framework handles all the execution details in a parallel environment allowing the analyst to focus on estimation. We design a novel asynchronous sampling estimator for parallel online aggregation over partitioned data. While achieving similar accuracy to existing estimators, our estimator is much more robust to node delays and failures. To verify the capabilities of the framework, we compare our estimator with two existing estimators by implementing and executing them in the framework. The results confirm the ability of the framework to execute the estimators without incurring any remarkable overhead and to provide tight confidence bounds early in the execution for highly selective queries, skewed data, and other pathological cases. The reason

for this is the extremely efficient tuple discovery mechanism that takes advantage of multi-node and multi-threaded parallelism.

We plan to address some of the limitations of the framework and extend its capabilities in future work. We have already started to investigate how to provide online estimates for asynchronous parallel joins when none of the two relations fits in memory. We plan to incorporate other estimation methods than sampling in the framework, for example Bayesian statistics and bootstrapping. Our long-term goal is to provide online estimates for any computation without incurring any overhead. We believe this is possible given the amount of parallelism available in modern processors.

Acknowledgments. This work was supported in part by a gift from LogicBlox.

References

1. Hadoop. `http://hadoop.apache.org/`. [Online; accessed July 2011].
2. TPC-H. `http://www.tpc.org/tpch/`. [Online; accessed February 2012].
3. F. Olken. Random Sampling from Databases. Ph.D. thesis, 1993. UC Berkeley.
4. S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12):1902–1905, 2012.
5. S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez. The DataPath System: a Data-Centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, pages 519–530, 2010.
6. R. Avnur, J. M. Hellerstein, B. Lo, C. Olston, B. Raman, V. Raman, T. Roth, and K. Wylie. CONTROL: Continuous Output and Navigation Technology with Refinement On-Line. In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data*, pages 567–569, 1998.
7. S. Chen, P. B. Gibbons, and S. Nath. PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, pages 147–158, 2010.
8. Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, pages 697–700, 2012.
9. W. G. Cochran. *Sampling Techniques*. Wiley, 1977.
10. S. Cohen. User-Defined Aggregate Functions: Bridging Theory and Practice. In *Proceedings of 2006 ACM SIGMOD International Conference on Management of Data*, pages 49–60, 2006.
11. T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of 2010 USENIX Conference on Networked Systems Design and Implementation*, pages 21–32, 2010.
12. G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.
13. A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-Charging Estimate Convergence in DBO. *PVLDB*, 2(1):419–430, 2009.
14. X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, pages 325–336, 2012.
15. M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of 2001 VLDB International Conference on Very Large Databases*, 2001.
16. P. J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *Proceedings of 1997 SSDBM International Conference on Scientific and Statistical Database Management*, pages 51–63, 1997.
17. P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of 1999 ACM SIGMOD International Conference on Management of Data*, pages 287–298, 1999.
18. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, pages 171–182, 1997.
19. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. *SIGMOD Rec.*, 26(2), 1997.

20. C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. In *Proceedings of 2007 ACM SIGMOD International Conference on Management of Data*, pages 725–736, 2007.
21. C. Jermaine, A. Dobra, S. Arumugam, S. Joshi, and A. Pol. The Sort-Merge-Shrink Join. *ACM TODS*, 31(4), 2006.
22. C. Jermaine, A. Dobra, A. Pol, and S. Joshi. Online Estimation for Subset-Based SQL Queries. In *Proceedings of 2005 VLDB International Conference on Very Large Databases*, pages 745–756, 2005.
23. N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 5(10):1028–1039, 2012.
24. G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A Scalable Hash Ripple Join Algorithm. In *Proceedings of 2002 ACM SIGMOD International Conference on Management of Data*, pages 252–262, 2002.
25. N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.
26. C. Qin, F. Rusu. PF-OLA: A High-Performance Framework for Parallel On-Line Aggregation. *CoRR*, abs/1206.0051, 2012.
27. C. Qin, F. Rusu. Sampling Estimators for Parallel Online Aggregation. In *Proceedings of 2013 BNCOD British National Conference on Databases*, pages 204–217, 2013.
28. C. Qin, F. Rusu. Parallel Online Aggregation in Action. In *Proceedings of 2013 SSDBM International Conference on Scientific and Statistical Database Management*, 2013.
29. L. A. Rowe and M. Stonebraker. The POSTGRES Data Model. In *Proceedings of 1987 VLDB International Conference on Very Large Databases*, pages 83–96, 1987.
30. F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *Operating Systems Review*, 46(1):12–18, 2012.
31. F. Rusu, F. Xu, L. L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The DBO Database System. In *Proceedings of 2008 ACM SIGMOD International Conference on Management of Data*, pages 1223–1226, 2008.
32. H. Wang and C. Zaniolo. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems. In *Proceedings of 2000 VLDB International Conference on Very Large Databases*, pages 166–175, 2000.
33. M. Wu and C. Jermaine. A Bayesian Method for Guessing the Extreme Values in a Data Set. In *Proceedings of 2007 VLDB International Conference on Very Large Databases*, pages 471–482, 2007.
34. S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan. Distributed Online Aggregation. *PVLDB*, 2(1):443–454, 2009.
35. S. Wu, B. C. Ooi, and K.-L. Tan. Continuous Sampling for Online Aggregation over Multiple Queries. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, pages 651–662, 2010.
36. F. Xu, C. Jermaine, and A. Dobra. Confidence Bounds for Sampling-Based GROUP BY Estimates. *ACM TODS*, 33(3), 2008.