Sampling Estimators for Parallel Online Aggregation

Chengjie Qin and Florin Rusu

University of California, Merced
cqin3@ucmerced.edu, frusu@ucmerced.edu

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. When coupled with parallel processing, this allows for the interactive data exploration of the largest datasets. In this paper, we identify the main functionality requirements of sampling-based parallel online aggregation—partial aggregation, parallel sampling, and estimation. We argue for overlapped online aggregation as the only scalable solution to combine computation and estimation. We analyze the properties of existent estimators and design a novel sampling-based estimator that is robust to node delay and failure. When executed over a massive 8TB TPC-H instance, the proposed 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 achieves linear scalability.

Keywords: parallel databases, estimation, sampling, online aggregation.

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 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 [1] aims at reducing the duration of the 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 a subsequent query.

Online aggregation in a centralized setting received a lot of attention since its introduction in the late nineties. The extension to parallel environments was mostly considered unnecessary – when considered, it was direct parallelization of serial algorithms – given the performance boost obtained in such systems by simply increasing the physical resources. With the unprecedented increase in data volumes and the proliferation of multi-core processors, parallel online aggregation becomes a necessary tool in the Big Data analytics landscape. It is the combination of parallel processing and estimation what truly makes interactive exploration of massive datasets feasible.

G. Gottlob et al. (Eds.): BNCOD 2013, LNCS 7968, pp. 204–217, 2013.

In this paper, we identify the main requirements for parallel online aggregation—partial aggregation, parallel sampling, and estimation. Partial aggregation requires the extraction of a snapshot of the system during processing. What data are included in the snapshot is the result of parallel sampling, while estimates and confidence bounds for the query result are computed from the extracted samples. Our specific contributions are as follows:

- We discuss in details each stage in the parallel online aggregation process.
- We analyze and thoroughly compare the existent parallel sampling estimators.
- We introduce a scalable sampling estimator which exhibits increased accuracy in the face of node delay and failure.
- We provide an implementation for the proposed estimator that confirms its accuracy even for extremely selective queries over a massive 8TB TPC-H instance.

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 focus on the computation of general SELECT-PROJECT-JOIN (SPJ) queries having the following SQL form:

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

where $\bullet$ is the concatenation operator, f is an arbitrary *associative decomposable aggregate function* [2] over the tuple created by concatenating t_1 and t_2, and P is some boolean predicate that can embed selection and join conditions. 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.

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.

Online aggregation. The idea in online aggregation is to compute only an estimate of the aggregate result based on a sample of the data [1]. In order to provide any useful information though, the estimate is required to be accurate and statistically significant. Different from one-time estimation [3] 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* [4,5] regard the choice of the sample size and reusing the work done across iterations.

An alternative that avoids these problems altogether is to completely *overlap query processing with estimation* [6,7]. 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.

3 Parallel Online Aggregation

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 that has to be analyzed in order to derive confidence bounds. In this section, we discuss in detail each of these aspects for the overlapped online aggregation approach.

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. Thus, an important decision that has to be taken when designing an online aggregation system is how to generate random samples.

Centralized sampling. According to the literature [8], 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 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.

Sampling synopses. It is important to make the distinction between the runtime sampling methods used in online aggregation and estimation based on static samples taken offline [3], 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 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. 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.

Global randomization. A somehow more complicated solution is to make sure that a tuple can reside at any position in any partition—*global randomization*. 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, each node partitions the local data into sets corresponding to all the other nodes in the environment. In the second stage, 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 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 virtually 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 [4] requires synchronization between all the sampling processes executed at partition level in order to guarantee that the same fraction of the data are sampled at each partition. To implement this strategy, serialized access to a common resource is required for each item processed. This results in unacceptable execution time increase when estimation is active.

In the second solution, which we name *multiple estimators*, 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 [10] follows this approach. The main problem with the multiple estimators strategy is that the final result computation and the estimation are separate 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. 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 4 we provide insights into the actual implementation, while in Section 5 we present experimental results to evaluate the accuracy of the estimator and the runtime performance of the estimation.

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.

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 SQL aggregate in Eq. 1 as follows:

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

where `f` and `P` are the aggregate function and the boolean predicate embedding selection and join conditions, respectively. 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 [11,12,13] 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. If the normality condition does not hold, more conservative distribution-independent confidence bounds can be derived using the Chebyshev-Chernoff inequalities, for example.

A closer look at the variance formula in Eq. 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*

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

is an unbiased estimator for the variance in Eq. 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 a parallel setting where data are partitioned across multiple processing nodes.

Before that though, we discuss on how to extend the generic estimator to `GROUP BY` and general `SPJ` queries. For `GROUP BY`, a pair of estimators X and $\text{Est}_{\text{Var}(X)}$ can be defined independently for each group. The only modification is that predicate `P` includes an additional selection condition corresponding to the group. A detailed analysis on how X and $\text{Est}_{\text{Var}(X)}$ can be extended to general `SPJ` queries is given in [11]. The main idea is to include the join condition in predicate `P` and take into consideration the effect it has on the two samples. We do not provide more details since the focus of this paper is on parallel versions of X and $\text{Est}_{\text{Var}(X)}$.

Single Estimator Sampling. When 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 [4] 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. 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 Eq. 2 can be directly applied without any modifications.

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 is defined for each partition as $X_i = \frac{|D_i|}{|S_i|} \sum_{s \in S_i, \mathtt{P}(s)} \mathtt{f}(s)$ based on the generic sampling estimator in Eq. 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 data randomization at each processing node is sufficient for the analysis to hold.

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. 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 [9]. This is not true in the general case though with the variance of the estimators being entirely determined by the samples at hand. In a highly asynchronous parallel setting, 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 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 the processing concludes.

4 Implementation

We implement the sampling estimators for online aggregation in GLADE [2,14], a parallel processing system optimized for the execution of associative-decomposable User-Defined Aggregates (UDA). In this section, we discuss the most significant extensions made to the GLADE framework in order to support online aggregation. Then, we present the implementation of the single estimator as an example UDA.

Extended UDA Interface. Table 1 summarizes the extended UDA interface we propose for parallel online aggregation. This interface abstracts aggregation and estimation

Table 1. Extended UDA interface

Method	Usage

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

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

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 GLADE. Thus, the user can focus only on estimation modeling.

The first 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. There are at least two methods that need to be added to the UDA interface—`EstTerminate` and `EstMerge`. `EstTerminate` computes a local estimator at each node. It is invoked after merging the local UDAs during the estimation process. `EstMerge` is called to put together in a single UDA the estimators computed at each node by `EstTerminate`. It is invoked with UDAs originating at different nodes. Notice that `EstTerminate` is an intra-node method while `EstMerge` is inter-node. It is possible to further separate the estimation from aggregate computation and have an intra-node `EstMerge` and an inter-node `EstTerminate`.

The second extension 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.

Example UDA. We present the UDA corresponding to the proposed asynchronous estimator for single-table aggregation – more diverse examples of higher complexity are presented in [15] – having the following SQL form:

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

which computes the SUM of function `f` applied to each tuple in table `TABLE` that satisfies condition `P`. It is straightforward to express this aggregate in UDA form. The `state` consists only of the running sum, initialized at zero. `Accumulate` updates the

Algorithm 1. *UDASum-SingleEstimator*

State: *sum*; *sumSq*; *count*

Init ()

1. $sum = 0$; $sumSq = 0$; $count = 0$

Accumulate (Tuple `t`)

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

Merge (UDASum $input_1$, UDASum $input_2$, UDASum $output$)

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

current sum with `f(t)` only for the tuples `t` satisfying the condition `P`, while `Merge` adds the states of the input UDAs and stores the result as the state of the output UDA.

UDASum-SingleEstimator implements the estimator we propose. No modifications to the UDA interface are required. Looking at the UDA 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`.

Parallel Online Aggregation in GLADE. At a high level, enhancing GLADE with online aggregation is just a matter of providing support for UDAs expressed using the extended UDA interface in Table 1 in order to extract a snapshot of the system state that can be used for estimation. While this is a good starting point, there are multiple aspects that require careful consideration. For instance, the system is expected to 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. Abstractly, this corresponds to executing two simultaneous UDA computations. Rather than treating actual computation and estimation as two separate UDAs, we group everything into a single UDA satisfying the extended interface. More details can be found in an extended version of the paper [15].

5 Empirical Evaluation

We present experiments that compare the asynchronous single estimator we propose in this paper and the multiple estimators approach. We evaluate the "time 'til utility" (TTU) [13] or convergence rate of the estimators and the scalability of the estimation process on a 9-node shared nothing cluster—one node is configured as the coordinator and the other 8 nodes are workers. The dataset used in our experiments is TPC-H scale **8,000** (***8TB***)—each node stores $1TB$. For more details on the experimental setup, we refer the reader to our extended report [15].

The aggregation task we consider is given by the following general `SPJ` query:

```
SELECT n_name, SUM(l_extendprice*(1-l_discount)*(1+l_tax))
FROM lineitem, supplier, nation
WHERE l_shipdate = 1993-02-26 AND l_quantity = 1 AND
 l_discount between [0.02,0.03] 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 as in the group-by case. This join strategy is common in parallel databases.

What is important to notice about this query is the extremely high selectivity of the selection predicates. Out of the 48×10^9 tuples in `lineitem`, only $35,000$ tuples are part of the result. These tuples are further partitioned by the `GROUP BY` clause such that the number of tuples in each group is around $1,500$. This corresponds to a selectivity of 29×10^{-7}—a veritable needle in the haystack query. Providing sampling estimates for so highly selective queries is a very challenging task.

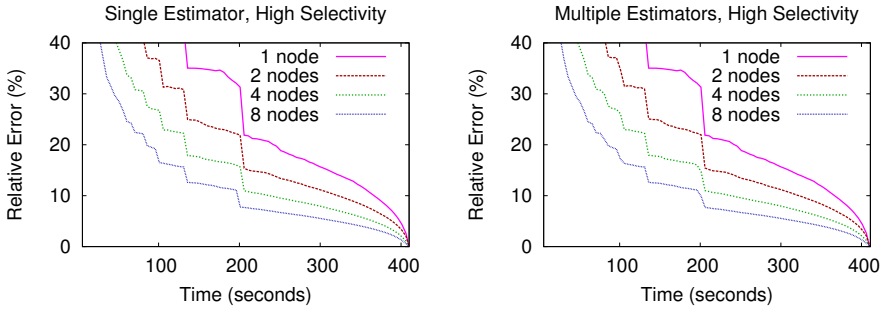

Fig. 1. Comparison between single estimator and multiple estimators. The plots print the results corresponding to the `PERU` group.

The results are depicted in Figure 1. As expected, the accuracy of the two estimators increases as more data are processed, converging on the correct result in the end. The effect of using a larger number of processing nodes is also clear. With 8 nodes more result tuples are discovered in the same amount of time, thus the better accuracy. Since the query takes the same time when proportionally more data and processing nodes are used, the scaleup of the entire process is also confirmed. What is truly remarkable though, is the reduced TTU even for this highly selective query. Essentially, the error is already under 10% when less than half of the data are processed. The reason for this is the effective tuple discovery process amplified by parallel processing.

When comparing the two estimators, there is no much difference—both in accuracy and in execution time. This confirms the effectiveness of the proposed estimator since the multiple estimators approach is known to have optimal accuracy in this particularly balanced scenario. It is also important to notice that the execution time is always limited by the available I/O throughput. The difference between the two estimators is clear when straggler nodes are present or when nodes die. Essentially, no estimate can be computed by the multiple estimators approach when any node dies. We refer the reader to the extended version of the paper [15] for experiments concerning the reliability of the estimators—and many other empirical evaluations.

6 Related Work

There is a plethora of work on online aggregation published in the database literature starting with the seminal paper by Hellerstein et al. [1]. We can broadly categorize this body of work into system design [16,6], online join algorithms [17,11,18], and methods to derive confidence bounds [17,11,12]. All of this work is targeted at single-node centralized environments.

The parallel online aggregation literature is not as rich though. We identified only three lines of research that are closely related to this paper. Luo et al. [10] extend the centralized ripple join algorithm [17] to a parallel setting. A stratified sampling estimator [9] is defined to compute the result estimate while confidence bounds cannot always be derived. This is similar to the multiple estimators approach. Wu et al. [4] extend online aggregation to distributed P2P networks. They introduce a synchronized sampling estimator over partitioned data that requires data movement from storage nodes to processing nodes. This corresponds to the synchronized single estimator solution.

The third piece of relevant work is online aggregation in Map-Reduce. In [19], stock Hadoop is extended with a mechanism to compute partial aggregates. In subsequent work [7], an estimation framework based on Bayesian statistics is proposed. BlinkDB [20] implements a multi-stage approximation mechanism based on precomputed sampling synopses of multiple sizes, while EARL [5] is an iterative online aggregation system that uses bootstrapping to produce multiple estimators from the same sample. Our focus is on sampling estimators for overlapped online aggregation. This is a more general problem that subsumes sampling synopses and estimators for iterative online aggregation.

7 Conclusions

We propose the combination of parallel processing and online aggregation as a feasible solution for Big Data analytics. We identify the main stages – partial aggregation, parallel sampling, and estimation – in the online aggregation process and discuss how they can be extended to a parallel environment. We design a scalable sampling-based estimator with increased accuracy in the face of node delay and failure. We implement the estimator in GLADE [2] – a highly-efficient parallel processing system – to confirm its accuracy even for extremely selective queries over a massive TPC-H 8TB instance.

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

References

1. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: SIGMOD (1997)
2. Rusu, F., Dobra, A.: GLADE: A Scalable Framework for Efficient Analytics. Operating Systems Review 46(1) (2012)
3. Cormode, G., Garofalakis, M.N., Haas, P.J., Jermaine, C.: Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. Foundations and Trends in Databases 4(1-3) (2012)
4. Wu, S., Jiang, S., Ooi, B.C., Tan, K.L.: Distributed Online Aggregation. PVLDB 2(1) (2009)
5. Laptev, N., Zeng, K., Zaniolo, C.: Early Accurate Results for Advanced Analytics on MapReduce. PVLDB 5(10) (2012)
6. Rusu, F., Xu, F., Perez, L.L., Wu, M., Jampani, R., Jermaine, C., Dobra, A.: The DBO Database System. In: SIGMOD (2008)
7. Pansare, N., Borkar, V.R., Jermaine, C., Condie, T.: Online Aggregation for Large MapReduce Jobs. PVLDB 4(11) (2011)
8. Olken, F.: Random Sampling from Databases. Ph.D. thesis, UC Berkeley (1993)
9. Cochran, W.G.: Sampling Techniques. Wiley (1977)
10. Luo, G., Ellmann, C.J., Haas, P.J., Naughton, J.F.: A Scalable Hash Ripple Join Algorithm. In: SIGMOD (2002)
11. Jermaine, C., Dobra, A., Arumugam, S., Joshi, S., Pol, A.: The Sort-Merge-Shrink Join. TODS 31(4) (2006)
12. Jermaine, C., Arumugam, S., Pol, A., Dobra, A.: Scalable Approximate Query Processing with the DBO Engine. In: SIGMOD (2007)
13. Dobra, A., Jermaine, C., Rusu, F., Xu, F.: Turbo-Charging Estimate Convergence in DBO. PVLDB 2(1) (2009)
14. Cheng, Y., Qin, C., Rusu, F.: GLADE: Big Data Analytics Made Easy. In: SIGMOD (2012)
15. Qin, C., Rusu, F.: PF-OLA: A High-Performance Framework for Parallel On-Line Aggregation. CoRR abs/1206.0051 (2012)
16. Avnur, R., Hellerstein, J.M., Lo, B., Olston, C., Raman, B., Raman, V., Roth, T., Wylie, K.: CONTROL: Continuous Output and Navigation Technology with Refinement On-Line. In: SIGMOD (1998)
17. Haas, P.J., Hellerstein, J.M.: Ripple Joins for Online Aggregation. In: SIGMOD (1999)
18. Chen, S., Gibbons, P.B., Nath, S.: PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. In: SIGMOD (2010)
19. Condie, T., Conway, N., Alvaro, P., Hellerstein, J.M., Elmeleegy, K., Sears, R.: MapReduce Online. In: NSDI (2010)
20. Agarwal, S., Panda, A., Mozafari, B., Iyer, A.P., Madden, S., Stoica, I.: Blink and It's Done: Interactive Queries on Very Large Data. PVLDB 5(12) (2012)