Bi-Level Online Aggregation on Raw Data

Yu Cheng*
Turn, Inc.
leo.cheng@turn.com

Weijie Zhao
University of California Merced
wzhao23@ucmerced.edu

Florin Rusu
University of California Merced
frusu@ucmerced.edu

ABSTRACT

In-situ processing has been proposed as a novel data exploration solution in many domains generating massive amounts of raw data, e.g., astronomy, since it provides immediate SQL querying over raw files. The performance of in-situ processing across a query workload is, however, limited by the speed of full scan, tokenizing, and parsing of the entire data. Online aggregation (OLA) has been introduced as an efficient method for data exploration that identifies uninteresting patterns faster by continuously estimating the result of a computation during the actual processing—the computation can be stopped as early as the estimate is accurate enough to be deemed uninteresting. However, existing OLA solutions have a high upfront cost of randomly shuffling and/or sampling the data.

In this paper, we present OLA-RAW, a bi-level sampling scheme for parallel online aggregation over raw data. Sampling in OLA-RAW is query-driven and performed exclusively in-situ during the runtime query execution, without data reorganization. This is realized by a novel resource-aware bi-level sampling algorithm that processes data in random chunks concurrently and determines adaptively the number of sampled tuples inside a chunk. In order to avoid the cost of repetitive conversion from raw data, OLA-RAW builds and maintains a memory-resident bi-level sample synopsis incrementally. We implement OLA-RAW inside a modern in-situ data processing system and evaluate its performance across several real and synthetic datasets and file formats. Our results show that OLA-RAW chooses the sampling plan that minimizes the execution time and guarantees the required accuracy for each query in a given workload. The end result is a focused data exploration process that avoids unnecessary work and discards uninteresting data.

CCS CONCEPTS

• **Information systems → Online analytical processing engines**; *DBMS engine architectures*; *Query operators*;

ACM Reference format:
Yu Cheng, Weijie Zhao, and Florin Rusu. 2017. Bi-Level Online Aggregation on Raw Data. In *Proceedings of SSDBM '17, Chicago, IL, USA, June 27-29, 2017,* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3085504.3085514

*Work done while a Ph.D. student at UC Merced.

SSDBM '17, June 27-29, 2017, Chicago, IL, USA

ACM ISBN 978-1-4503-5282-6/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3085504.3085514

1 INTRODUCTION

In the era of data deluge, massive amounts of *raw data* are generated at an unprecedented scale by mobile applications, sensors, and scientific experiments. The vast majority of these read-only data are stored as application-specific files containing millions – if not billions – of records. *Data exploration* is the initial step in extracting knowledge from these data. Aggregate statistics are computed in order to assess the quality of the raw data, before a thorough investigation on transformed data is performed. The main goal of data exploration is to determine if the time-consuming data transformation and in-depth analysis are necessary. Thus, data exploration does not have to be exact. As long as accurate *estimates* that guide the decision process are generated, its goal is achieved. This allows for an extensive set of optimization strategies that reduce I/O and CPU utilization to be employed. However, if the detailed analysis is triggered, the work performed during exploration should allow for *incremental* extension. In order to illustrate these concepts, we provide an example from a real application in astronomy.

Motivating example. The Palomar Transient Factory[1] (PTF) project [33] aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. A list of potential transients – or candidates – is extracted from the images taken by the telescope during a night. They are stored as a table in one or more FITS[2] files. The initial stage in the identification process is to execute a series of aggregate queries over the batch of extracted candidates. This corresponds to data exploration. The general SQL form of the queries is:

```
SELECT AGGREGATE(expression) AS agg
FROM candidate
WHERE predicate
HAVING agg < threshold
```

where *AGGREGATE* is SUM, COUNT, or AVERAGE and *threshold* is a verification parameter. These queries check certain statistical properties of the entire batch and are executed in sequence—a query is executed only if all the previous queries are satisfied. If the candidate batch passes the verification criteria, an in-depth analysis is performed for individual candidates. The entire process – verification and in-depth analysis – is executed by querying a PostgreSQL[3] database—only after the candidates are loaded from the original FITS files. This workflow is highly inefficient for two reasons. First, the verification cannot start until data are loaded. Second, if the batch does not pass the verification, both the time spent for loading and the storage used for replication are wasted.

Raw data processing. To reduce the high upfront database loading cost, multiple raw data processing systems have been recently introduced [1, 4, 9, 22, 23, 36]. They are extensions of the external

[1] www.astro.caltech.edu/ptf/
[2] http://heasarc.gsfc.nasa.gov/fitsio/
[3] http://www.postgresql.org/

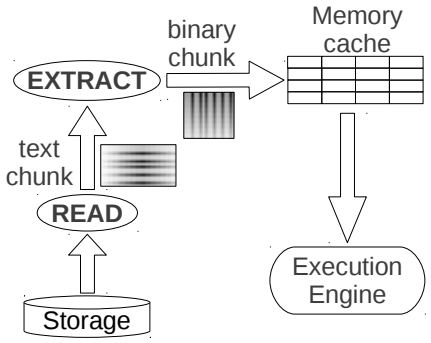

Figure 1: Raw data processing.

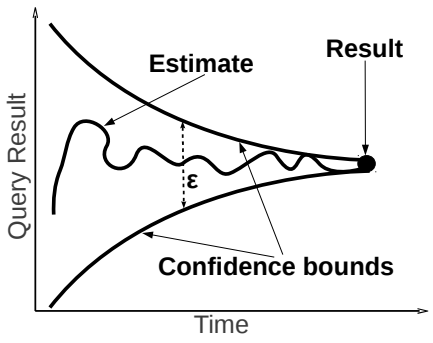

Figure 2: Online aggregation.

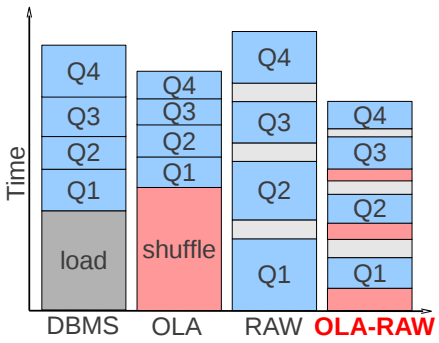

Figure 3: OLA-RAW approach.

table mechanism supported by standard database servers. These systems execute SQL queries directly over raw data while optimizing the conversion process into the format required by the query engine (Figure 1). This eliminates loading and provides instant access to data—verification can start immediately in our example. However, since verification consists of more than a single query, the overall time incurred by raw data processing can be larger than in a database because full access to the raw data is required for every query. Several systems [1, 4, 9] provide a dynamic tradeoff between the time to access data and the query execution time by adaptively loading a portion of the data during processing. This allows for gradually improved query execution times while reducing the amount of storage for replication. However, these systems are data-agnostic and cannot identify uninteresting patterns early in the processing. This results in wasted CPU and storage resources.

Online aggregation. Since the goal of data exploration – batch verification in our example – is only to determine if the individual candidate in-depth analysis is necessary, it is not mandatory to evaluate each query in the sequence to completion. As early as the relationship between the aggregate and the verification threshold can be accurately inferred, the query can be stopped. This relationship can be determined by using only an estimate of the aggregate. If the aggregate agg – or its estimate – is larger than the threshold, the verification fails and no in-depth analysis is required. Otherwise, we can proceed to the subsequent query in the verification. Online aggregation (OLA) [14, 21, 25] provides a sound framework to reason about the aggregate estimation involved in verification. The main idea in OLA is to estimate the query result based on a sample of the data. In addition to the estimator, OLA defines a principled method to derive confidence bounds that permit the correct identification of the relationship with the threshold. The estimator and the bounds are computed from a sample much smaller in size than the overall dataset, thus reducing the execution time of a verification query tremendously (Figure 2). Data shuffling [39, 40, 46] is the standard procedure to extract samples of increasing size from a dataset. Shuffling generates a permutation of the data as a preprocessing step such that a runtime sequential scan results in random samples of increasing size. However, shuffling creates a secondary copy of the data and incurs significant processing time.

Problem & approach. At high-level, our objective is to *optimally execute exploration over raw data in a shared-memory multicore environment where I/O operations are overlapped with extraction and several chunks can be processed concurrently while minimizing resource utilization.* In our example, this corresponds to low execution time for verification without incurring any loading cost.

Our approach is to seamlessly integrate online aggregation into raw data processing such that we cumulate their benefits. Figure 3 illustrates the intuition behind the proposed OLA-RAW solution with respect to standard database processing (DBMS), online aggregation (OLA), and raw data processing with adaptive loading (RAW), respectively. Similar to RAW, OLA-RAW distributes data loading across the query workload. Notice, though, that loading in RAW – and, by extension, in OLA-RAW – corresponds to caching data in memory, not necessarily materializing on secondary storage. The same idea is extended to shuffling. Instead of randomly permuting all the data before performing online aggregation, OLA-RAW partitions shuffling across the queries in the workload. Moreover, loading and shuffling are combined incrementally such that loaded data do not require further shuffling. Essentially, *OLA-RAW provides a resource-aware parallel mechanism to adaptively extract and incrementally maintain samples from raw data.* The end goal is to reduce the high upfront cost of loading (DBMS) and shuffling (OLA), and to minimize the amount of data accessed by RAW, as long as estimates are accepted by the user.

Challenges. The realization of OLA-RAW poses a series of difficult challenges. First and foremost, an efficient sampling mechanism targeted at raw data has to be devised. The sampling mechanism has to work in-place, over data in the original format. It has to minimize the amount of raw data read and/or extracted into the processing representation since these are the fundamental limitations of raw data processing. Given our focus on parallel processing, the sampling mechanism has to cope with the so called "inspection paradox" [38]. Since the estimate is correlated with the extraction time, the order in which chunks are considered has to be the same with the extraction scheduling order. The second challenge is defining and analyzing estimators for the sampling mechanism. In order to be amenable to online aggregation, the estimators have to be integrated in the sampling mechanism and they have to support incremental computation over samples of increasing size. A third challenge corresponds to the incremental maintenance of samples. Since extracting samples from raw data is expensive, a mechanism that preserves them in memory for further use in subsequent queries and maintains them incrementally is necessary. This has to be realized efficiently—the goal is to compute the estimate as fast

as possible, not to maintain the sample. From an implementation perspective, the integration of online aggregation into a resource-aware raw data processing system is challenging because of the complex interactions between I/O, extraction, and sampling.

Contributions. The main contribution of this paper is a novel bi-level sampling scheme for OLA-RAW that addresses the aforementioned challenges. OLA-RAW sampling is query-driven and performed exclusively in-situ during query execution, without data reorganization. In order to avoid the expensive conversion cost, OLA-RAW builds and maintains incrementally a memory-resident bi-level sample synopsis. These are achieved through a series of technical contributions detailed in the following:

- We define online aggregation over raw data in a multi-core shared-memory setting (Section 2).
- We introduce a novel parallel bi-level sampling scheme that supports continuous estimation – not only at chunk boundaries – during the online aggregation process (Section 3).
- We devise a resource-aware policy to determine the optimal chunk sample size for bi-level sampling (Section 4).
- We design a memory-resident bi-level sample synopsis that is built and maintained incrementally following a variance-driven strategy (Section 5).

We implement OLA-RAW inside a state-of-the-art in-situ data processing system and evaluate its performance across several real and synthetic datasets and file formats. We investigate the importance of each OLA-RAW component as well as the overall solution. Our results (Section 6) show that OLA-RAW chooses the sampling plan that minimizes the execution time and guarantees the required accuracy for each query in a given workload.

2 PRELIMINARIES

In this section, we introduce query processing over raw data and online aggregation, respectively. We define the online aggregation over raw data problem and identify the challenges that have to be addressed by a coherent solution that integrates the two.

Parallel raw data processing. Raw data processing is depicted in Figure 1. The input to the process is a raw file from a non-volatile storage device, e.g., disk or SSD, a schema that can include optional attributes, and a procedure to extract tuples with the given schema from the raw file. The output is a tuple representation that can be processed by the query engine and, possibly, is cached in memory. In the `READ` stage, data are read from the original raw file, chunk-by-chunk, using the file system's functionality. A chunk contains multiple records and represents the unit of processing. Without additional information about the structure or the content – stored inside the file or in some external structure – the entire file has to be read the first time it is accessed. `EXTRACT` transforms tuples from raw format into the processing representation based on the schema provided and using the extraction procedure given as input to the process. There are two main tasks in `EXTRACT`. The first is to identify the schema attributes and output a vector containing the starting position for every attribute in the tuple—or a subset, if the query does not access all the attributes. Second, attributes are converted from the raw format to their corresponding binary type and mapped to the processing representation of the tuple—the record in a row-store, or the array in column-stores, respectively. At the end of `EXTRACT`, data are loaded in memory and ready for query processing. In this paper, we consider parallel raw data processing in the context of the SCANRAW operator [9, 10] and NoDB [4]. SCANRAW overlaps the I/O operations with `EXTRACT` over multiple chunks in a super-scalar pipeline architecture, i.e., multiple chunks are extracted concurrently. NoDB caches binary chunks in memory to avoid subsequent extraction.

Online aggregation on raw data. We consider online aggregation over a table T stored in any sequential raw format, e.g., CSV, JSON, or FITS, and general aggregate queries of the form:

```
SELECT AGGREGATE(expression)
FROM T
WHERE predicate
```

where *AGGREGATE* is one of SUM, COUNT, or AVERAGE and *expression* is a numeric expression, such as $T.a$ or $(T.a - T.b)^2$, that involves one or more columns of T. These aggregation functions are the most commonly used in practice. Online aggregation over a single raw data source is a fundamental problem arising not only in queries that explicitly involve a single table – this is the standard scenario in raw data processing – but also in queries on "star" schemas that consist of a massive "fact" table – which is sampled – and many smaller "dimension" tables—which are cached in memory. In general, the proposed methods apply to queries that involve joins between multiple tables, provided that sampling is performed on exactly one raw data source and each join attribute is a foreign key. *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 the queries simultaneously. A group-specific version of the *predicate* that accepts only tuples from that particular group is required for each separate query.

In addition to the query, an online aggregation user is typically required to specify two parameters. The *accuracy* ϵ determines when the query can be stopped. Different from one-time estimation [14, 17] that might produce inaccurate estimates not satisfying a given ϵ, OLA 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. As more data are processed towards computing the result, the accuracy of the estimator improves accordingly (Figure 2). The *estimation time interval* δ specifies how often the estimate and corresponding confidence bounds are computed. The user can decide to stop the query at any time – even when the accuracy is not satisfied – based on the returned estimate and confidence bounds.

Extracting random samples of increasing size at runtime is a complex time-consuming process [37]. This is the reason why existing procedures require a certain level of preprocessing. In offline sampling, a series of samples with progressively increasing size – the largest one being the entire dataset – are taken, e.g., BlinkDB [2]. These samples are stored and processed as independent entities. Offline tuple-based shuffling [39, 40, 46] guarantees that a runtime sequential scan produces samples of increasing size. In addition to preprocessing time, the offline methods incur a heavy storage overhead. These are in contrast with the requirements of in-situ data processing. Chunk-level [14], i.e., block-level [38] or cluster [41], sampling samples over the chunk space instead of the tuple space.

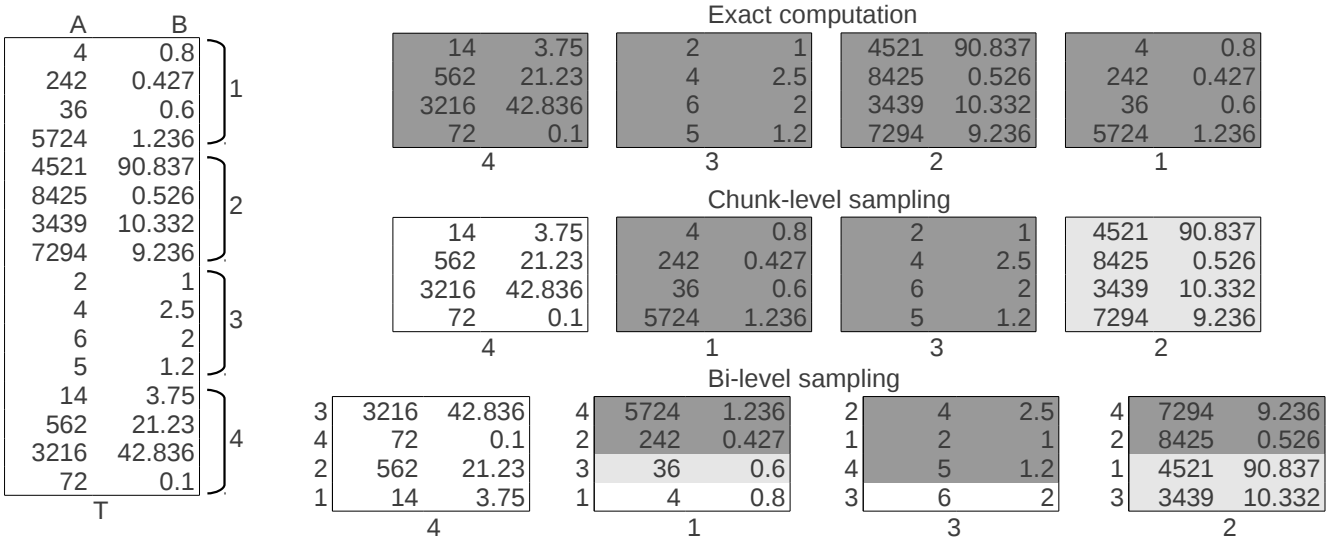

Figure 4: Sampling strategies for online aggregation over raw data. The dark gray portions correspond to data included in estimation. The light gray portions correspond to data under processing, but not yet included in estimation. The white portions correspond to unprocessed data. Chunks are scheduled for processing from right to left.

This can be done efficiently online – randomly permute the processing order of chunks – without any preprocessing. However, chunk-level sampling incurs a higher processing cost because all the tuples inside a chunk have to be included in estimation. While this may be irrelevant for database processing, it is of great importance for in-situ processing due to the EXTRACT stage.

In this paper, we consider chunk-level sampling in the context of parallel raw data processing, specifically the SCANRAW operator. This creates problems because the random chunk order interacts with parallel processing. This, in turn, can trigger the inspection paradox which makes sampling-based estimation impossible. The only solution that addresses this problem in a distributed MapReduce setting is given in [38]. It defines a multivariate distribution that incorporates several timing parameters in addition to the aggregate chunk value. Since we focus on multi-thread parallelism in a shared-memory setting, the timing parameters are too similar to have a discriminative effect. Moreover, we can take advantage of the centralized shared-memory environment to eliminate the inspection paradox without expensive distributed synchronization.

3 BI-LEVEL SAMPLING

In this section, we introduce a novel bi-level sampling scheme for parallel online aggregation. The proposed scheme differs significantly from bi-level Bernoulli sampling [20] in which the goal is to extract a one-time Bernoulli sample with rate q. Our objective is to design an incremental sampling procedure that supports adaptive size increase in order to achieve continuous accuracy improvement. Bi-level sampling combines chunk and tuple-level sampling optimally. Details on these primitive sample procedures are included in the extended version [11].

We propose the following bi-level sampling scheme. Chunks are read in a predetermined random order similar to chunk-level sampling. However, instead of aggregating all the tuples in the chunk into a single value that becomes a surrogate for the chunk, a secondary sampling process is performed over the tuples in the chunk. This is realized by randomly shuffling the order in which tuples are extracted. Independent orders are used across chunks. Since this is done in memory, no significant overhead is incurred. Figure 4 depicts the entire procedure.

At each step during the process, the set of sampled tuples correspond to a bi-level – or two-stage – sample without replacement [12, 41]. This provides more flexibility over chunk-level sampling since estimates can be computed at any point in the process—not only at chunk boundaries. Moreover, the sample size can be increased either by including more chunks or more tuples inside a chunk. It is important to notice that bi-level sampling degenerates to chunk-level sampling when all the tuples inside a chunk are included in the sample. This is likely to happen when there is high variability inside a chunk. When tuples inside a chunk are similar, however, bi-level sampling allows for the chunk to be represented by a much smaller number of tuples. This has the potential to dramatically reduce the cost of EXTRACT in raw data processing. The downside of bi-level sampling is the larger number of parameters. In addition to the number of chunks, the number of sampled tuples inside each chunk has to be specified. However, these are determined dynamically in online aggregation.

3.1 Parallel Procedure

As in the case of chunk-level sampling, when bi-level sampling is applied to parallel online aggregation over raw data, the inspection paradox can invalidate the entire sampling procedure. The impact of the inspection paradox is further aggravated by the different number of tuples extracted across chunks. For example, consider a highly-variable chunk followed by a uniform one (chunk 2 and 3

in Figure 4). Although chunk 2 is scheduled before chunk 3, fewer tuples from chunk 3 have to be extracted and included in estimation. As a result, chunk 3 – even chunk 1 in Figure 4 – is extracted before chunk 2. In the case of chunk-level sampling, an estimate can be generated only after chunk 2 finishes—the estimate includes chunk 3 and 1, as well. While exactly the same conditions apply to bi-level sampling, the secondary sampling process inside chunks provides additional opportunities not to delay estimation—an essential requirement in online aggregation.

A major contribution of this paper is a novel solution for continuous estimation. We devise a mechanism that enforces the existence of samples from all the chunks in EXTRACT at any estimation time interval δ. Each EXTRACT thread is configured with a timing parameter t^{eval} that specifies when samples from the chunk have to be produced. This can happen multiple times during the execution of EXTRACT. Since chunks are scheduled for extraction sequentially, this guarantees that samples are extracted in order. The number of tuples included in the sample, however, can vary based on the properties of the chunk. This is illustrated in Figure 4 where 3 tuples are sampled from chunk 3, while only 2 tuples from chunk 2 and 1, respectively. As long as the timing parameter t^{eval} is smaller than δ, improved estimates can be generated. t^{eval} is – in a sense – related to the variables t^{sch} and t^{proc} in [38]. However, instead of including it in the estimation snapshot, we use timing to enforce the bi-level sampling process and its corresponding estimation. The overhead incurred by the timing mechanism is minimal and reduces to inspecting a timer after groups of several tuples are extracted.

3.2 Estimation

We focus on the estimator for the SUM aggregate. COUNT is identical to SUM with *expression* = 1. As shown in [20], only minor modifications have to be made for complex aggregates such as AVERAGE, VARIANCE, or standard deviation. The notation used in our analysis is shown in Table 1. It adapts the notation for Bernoulli sampling used in [20] to sampling without replacement.

Let $\tau = \sum_{j \in U} y_j = \sum_{i \in T} x_i$ denote the true result of the query. In order to define an unbiased estimator for τ, we first introduce an estimator for y_j, the sum of the expression values in chunk j. This is the standard estimator for tuple-level sampling without replacement restricted to a chunk, i.e., $\widehat{y}_j = \frac{M_j}{m_j} y'_j$. It is well-known that $\widehat{y}_j$ is an unbiased estimator. The unbiased estimator for τ combines the chunk estimators in a standard sampling without replacement estimator over the chunks:

$$\widehat{\tau} = \frac{N}{n} \sum_{j=1}^{n} \widehat{y}_j = \frac{N}{n} \sum_{j=1}^{n} \frac{M_j}{m_j} \sum_{i \in C'_j} x_i \tag{1}$$

Confidence bounds are the backbone of online aggregation since they characterize the accuracy of the estimator. This requires the computation of the variance and the definition of a variance estimator based on the samples. By assuming normality based on the Central Limit Theorem [12], confidence bounds are computed from the cumulative distribution function (cdf) of the normal distribution. We provide the formulae for the bi-level sampling variance and a corresponding unbiased estimator in the following theorems which are based on [41].

Symbol	Meaning
T	Set of tuples in table
U	Set of chunks in table
C_j	Set of tuples on chunk j
T'	Set of tuples in sample
U'	Set of chunks in sample
C'_j	Set of tuples on chunk j that are in sample
$M = \lvert T \rvert$	Number of tuples in table
$N = \lvert U \rvert$	Number of chunks in table
$M_j = \lvert C_j \rvert$	Number of tuples on chunk j
$m = \lvert T' \rvert$	Number of tuples in sample
$n = \lvert U' \rvert$	Number of chunks in sample
$m_j = \lvert C'_j \rvert$	Number of tuples in sample of chunk j
x_i	Value of *expression* for the i^{th} tuple in table ($x_i = 0$ if tuple i fails to satisfy *predicate*)
$y_j = \sum_{i \in C_j} x_i$	Sum of x_i values on chunk j
$y'_j = \sum_{i \in C'_j} x_i$	Sum of x_i values in sample of chunk j
$y''_j = \sum_{i \in C'_j} x_i^2$	Sum of x_i^2 values in sample of chunk j

Table 1: Notation for bi-level sampling used in the paper.

Theorem 1. *The variance of the bi-level sampling estimator $\widehat{\tau}$ defined in Eq.* (1) *is given by:*

$$Var(\widehat{\tau}) = \frac{N}{N-1} \frac{N-n}{n} \sum_{j=1}^{N} \left(y_j - \frac{\sum_{i \in T} x_i}{N} \right)^2 + \frac{N}{n} \sum_{j=1}^{N} \left[\frac{M_j}{M_j - 1} \frac{M_j - m_j}{m_j} \sum_{i \in C_j} \left(x_i - \frac{y_j}{M_j} \right)^2 \right] \tag{2}$$

There are two distinct terms in the variance formula—the first for the variance between chunks and the second for the variance inside each chunk. The variance between chunks measures how a chunk deviates from the average across all the chunks. The variance inside a chunk computes the deviation of a tuple from the average value of the chunk. The sampling without replacement nature of the process is reflected in the scaling factors of the two terms. The more chunks are sampled, the smaller the scaling factor, i.e., $\frac{N-n}{n}$. The same applies to the number of tuples sampled inside a chunk. In the extreme case when all the chunks are sampled, the variance across chunks vanishes. This corresponds to stratified sampling. If all the tuples inside a sampled chunk are included, the variance inside that chunk reduces to zero. By ignoring the terms corresponding to non-sampled chunks, the resulting variance corresponds to the chunk-level sampling variance—as well as the estimator τ.

Theorem 2. *An unbiased estimator for the variance of bi-level sampling is given by:*

$$\widehat{Var(\widehat{\tau})} = \frac{N}{n} \frac{N-n}{n-1} \sum_{j=1}^{n} \left(\frac{M_j}{m_j} y'_j - \frac{\sum_{j'=1}^{n} \frac{M_j}{m_j} y'_{j'}}{n} \right)^2 + \frac{N}{n} \sum_{j=1}^{n} \left[\frac{M_j}{m_j} \frac{M_j - m_j}{m_j - 1} \sum_{i \in C'_j} \left(x_i - \frac{y'_j}{m_j} \right)^2 \right] \tag{3}$$

In order to evaluate the query estimator and estimate its variance, a series of chunk-level statistics have to be computed. They include m_j, y'_j, and y''_j. These quantities require minimum space and processing overhead beyond what is required by the actual query. Additionally, the total number of tuples and the number of tuples in each chunk have to be known. This is problematic for raw data in general. However, since most textual formats store a tuple per line, e.g., CSV and JSON, these values can be easily obtained by a simple execution of the command `wc -l`. Other formats, e.g., HDF5 and FITS, store these values in the file metadata.

4 OLA-RAW BI-LEVEL SAMPLING

Given a bi-level sample extracted from raw data, Theorem 2 allows us to derive confidence bounds for an aggregate query. However, the theorem does not specify what is the optimal sampling procedure. In this section, we investigate how to extract a bi-level sample that is optimal for online aggregation over raw data. We define this problem formally and first introduce a solution that guarantees convergence to the required accuracy in one pass over the data. Then, we design a novel resource-aware parallel sampling procedure that allocates the system resources optimally in order to achieve the desired accuracy.

4.1 Holistic Sampling

We start with a straightforward realization of the bi-level sampling procedure that does not consider how much to sample from a chunk—it samples the entire chunk. However, in order to avoid the inspection paradox and to reduce the interval between estimations, samples are extracted from each chunk at t^{eval} time intervals. We emphasize that this is not chunk-level sampling because any subset of the chunk represents a sample—including the complete chunk. By grouping the samples from all the chunks together, a bi-level sample is generated and an estimate and corresponding confidence bounds can be computed. If the required accuracy ϵ is satisfied, the query can be stopped. Otherwise, the exact answer is obtained if the query is executed to completion.

4.2 Optimization Formulation

The optimization problem for online aggregation over raw data is to minimize the query processing cost under the constraint of achieving the specified accuracy ϵ. We introduce a cost model for bi-level sampling starting from the cost model for parallel in-situ processing over raw data given in [49]. In this model, the query processing cost is defined as the maximum between the read I/O time $T_{I/O}$ and the CPU extraction time with P worker threads T_{CPU}—`READ` and `EXTRACT` are overlapped. In the case of bi-level sampling, $T_{I/O}$ is a linear function of the number of sampled chunks n, i.e., $T_{I/O} \approx n$, while T_{CPU} is a linear function of the number of sampled tuples inside the chunk, i.e., $T_{CPU} \approx \frac{1}{P} \cdot \sum_{j=1}^{n} m_j$. The constraint on the accuracy ϵ can be written as an inequality between the variance estimator $\widehat{\text{Var}}(\widehat{\tau})$ and a maximum variance Var_{max} derived from ϵ. Based on these, the bi-level sampling for online aggregation over raw data optimization problem can be expressed as follows:

$$\begin{aligned} &\text{minimize} \quad \max\left\{T_{I/O}(n), T_{CPU}(\overline{m_j})\right\} \text{ subject to} \\ &\text{constraint} \quad \widehat{\text{Var}}(\widehat{\tau})(n, \overline{m_j}) \leq Var_{max} \end{aligned} \tag{4}$$

where the variables to be optimized are n and $\overline{m_j}$, $j \in \{1, \ldots, n\}$. Closed-form formulae for sequential bi-level Bernoulli sampling are derived in [20]. They are based on the notion of chunk heterogeneity index which measures the variability of values within a chunk relative to the variability of values between chunks. These formulae cannot be extended to our max objective function. Moreover, computing the chunk heterogeneity index over raw data in exploration tasks defeats the purpose of in-situ processing. The algorithms proposed in [20] require the existence of a pilot sample or of complex chunk-level statistics such as the number of distinct values and the variance. These are not available for raw data.

4.3 Single-Pass Sampling

Since the variables in the optimization formulation cannot be computed offline, the strategy used in online aggregation is to execute the bi-level sampling process and check the constraint at runtime. However, this does not simplify the problem since we have to decide what is better: sample more chunks, i.e., increase n? or sample more tuples from the current chunk, i.e., increase m_j? While the variance decreases in both cases, we cannot quantify the overall impact on the objective function.

The solution we propose is to set the number of sampled chunks n and solve the optimization formulation only with m_j variables. The objective function reduces to minimizing the extraction time $T_{CPU}(\overline{m_j})$, i.e., the number of extracted tuples. We set $n = N$ because this eliminates the term corresponding to the variance between chunks in Eq. (3) while preserving the bi-level nature of the sampling process. Even with this simplification, we cannot derive closed-form formulae for the m_j variables without having knowledge of the chunk heterogeneity index. In this situation, our strategy is to decompose the optimization formulation into separate problems for each chunk. These problems have the potential to become independent because of the simplification applied to the variance. For this to be the case, though, we have to identify Var^j_{max} values for each chunk that guarantee the global constraint is satisfied if the constraints at chunk-level are satisfied. Since the same sampling without replacement process is performed inside a chunk, the local constraints can be written as:

$$\frac{M_j}{m_j} \frac{M_j - m_j}{m_j - 1} \sum_{i \in C'_j} \left(x_i - \frac{y'_j}{m_j}\right)^2 \leq Var^j_{max} \tag{5}$$

We obtain the relationship $\sum_{j=1}^{N} Var^j_{max} \leq Var_{max}$ between Var^j_{max} and Var_{max} by summing up the local constraints and enforcing the global constraint. While it is possible to choose the values of Var^j_{max} based on chunk properties, a simpler solution is to equally divide Var_{max} across the N chunks, i.e., $Var^j_{max} = \frac{1}{N} Var_{max}$, $\forall j$. This constant solution guarantees that more tuples are sampled from chunks with higher variability and less from chunks that are homogeneous. Since the input to our problem is ϵ rather than Var_{max}, we determine the corresponding values ϵ^j that satisfy the modified optimization formulation. They are given in Theorem 3 which is proven in the extended version [11].

Theorem 3. *Given an aggregate query and an accuracy ϵ, a bi-level sampling procedure in which the estimate of each chunk aggregate satisfies accuracy $\epsilon^j = \epsilon, \forall j$, produces a global estimate that satisfies accuracy ϵ when all the chunks are sampled.*

Theorem 3 provides an algorithm to independently sample across chunks, thus allowing for maximum concurrency. It guarantees that – in the worst scenario when all the chunks are sampled – the aggregate estimate meets the required accuracy. It is important to emphasize that an estimate can be computed at any instant during the process and the accuracy can be met before all the chunks are considered. This can be done exclusively from the raw data—without knowledge of any other statistical information beyond simple chunk-level tuple counts. Moreover, this worst-case algorithm provides improvements over parallel external tables – the standard mechanism for raw data processing – in terms of the number of tuples extracted.

4.4 Resource-Aware Sampling

The single-pass sampling strategy guarantees that the required accuracy is met after a pass over the data. While theoretically sound for a worst-case scenario, it does not take into account the runtime conditions. As soon as the local accuracy is satisfied, sampling for the chunk is terminated, even though computational resources may be available. As a result, the opportunity to further reduce the total variance of the estimator is missed. We address this shortcoming by introducing a novel bi-level sampling strategy that dynamically and adaptively determines how much to sample from a chunk by continuously monitoring the resource utilization of the system. While the original goal of single-pass sampling is preserved, we aim to maximize the chunk-level sample size without decreasing the rate at which chunks are processed.

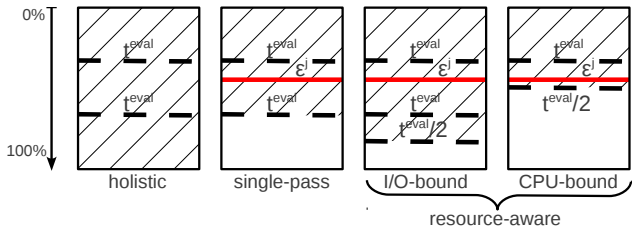

Figure 5: Bi-level sampling strategies in OLA-RAW.

In resource-aware sampling, the decision of when to stop sampling from a chunk is based on the availability of system resources – I/O bandwidth and CPU threads – in addition to the chunk accuracy. This requires continuous system monitoring at runtime. We monitor the buffer into which chunks are stored before EXTRACT and the pool of threads available to EXTRACT, each time an estimate is generated from a chunk, i.e., t^{eval}. The cost of monitoring does not incur significant overhead and it allows for the early detection of changes in the workload. As long as the number of threads is larger than the number of chunks in the buffer, processing is I/O-bound. Otherwise, it is CPU-bound. Since the objective function is the maximum of the two, we take different decisions in each case. They are depicted in Figure 5. In the case of I/O-bound in-situ processing, t^{eval} is halved only after the local chunk accuracy ϵ^j is achieved. For CPU-bound processing, this is done immediately after the first chunk estimate is generated. The rationale for these decisions is as follows. For I/O-bound processing, our goal is to end a chunk as soon as another chunk becomes available and there are no threads in the pool. In the case of CPU-bound tasks, the goal is to finalize a chunk as soon as the accuracy is met. By decreasing t^{eval}, we increase the monitoring frequency in the hope that we detect the triggering factor – accuracy or resource utilization – as early as possible.

The timing parameter t^{eval} plays a central role in the resource-aware parallel sampling procedure proposed in this paper. Its value controls the frequency at which chunk-level estimates are produced and the monitoring interval. Moreover, t^{eval} is shared across the EXTRACT threads in order to avoid the inspection paradox. Thus, determining the appropriate values for t^{eval} is of major importance. When processing starts, we assign t^{eval} a lower bound value, e.g., $1\,ms$. t^{eval} cannot decrease below this value at any time. The first several chunks are extracted using the initial lower bound value because CPU resources are plentiful—it takes some time until the EXTRACT pipeline is filled. During this calibration process, our goal is to accurately determine the t^{eval} value at which the chunk accuracy is achieved. We set t^{eval} to the calibration average in order to reduce the number of monitoring operations. The calibration continues during the entire processing and t^{eval} is updated with the current running average. A clear upper bound for t^{eval} is δ—the time interval at which results have to be presented to the user. Values larger than δ introduce hiccups in estimation. Another upper bound is represented by the time to process an entire chunk—this time is also monitored. In this case, the process degenerates to chunk-level sampling. Thus, the minimum of the two is taken as the upper bound. The monitoring in resource-constrained situations is done using an exponential decay scheme. The value of t^{eval} is repeatedly halved to increase the monitoring frequency. Nonetheless, the lower bound is enforced. This scheme is especially important when the accuracy ϵ is always satisfied at the first monitoring and the processing is CPU-bound since it forces the global t^{eval} to drop.

5 BI-LEVEL SAMPLE SYNOPSIS

Extracting a bi-level sample from raw data is expensive. Since data exploration requires a sequence of queries, this process becomes the bottleneck if executed from scratch for each query. It is important to notice that extracting the sample once and using it for multiple queries is not possible in our scenario because sampling is driven by the current query. It is very likely that the sample corresponding to a query cannot be used for another query, e.g., it does not include all the required columns. In this case, access to the raw data is necessary in order to extract the missing columns. This requires complete resampling and provides little opportunity for improvement. The more interesting case is when the same sample can be used across queries, however, the required accuracy for the new query cannot be met. This can happen because, for example, the user-specified accuracy ϵ increases, the new query is more selective, or the new aggregate is different. In this situation, the goal is to preserve and incrementally maintain the sample.

We propose a novel *memory-resident bi-level sample synopsis* that is built and maintained based on the query workload. Since the size of the synopsis is bound by a specified memory budget B, the synopsis is not necessarily identical to the bi-level sample extracted from the raw data and used for estimation. The synopsis is built based on a query specified by the user, i.e., origin query, and is updated by each subsequent query—only if the query cannot be answered exclusively using the synopsis. A query that cannot take advantage of the synopsis triggers a complete rebuild automatically.

Several aspects underlie the novelty of the bi-level sample synopsis with respect to other sampling strategies used in online aggregation. First, the proposed synopsis is computed entirely at runtime and does not require any preprocessing, e.g., offline shuffling or sampling. Second, the synopsis is driven by the queries. There is no generic process to generate universal samples applicable to any query. Third, the synopsis caches in memory a subsample of the data to be used directly by subsequent queries. No other solution considers a query sequence. Each query starts sampling the data from scratch. While the idea of caching extracted data in memory is similar to NoDB [4], the proposed bi-level sample synopsis does not require caching full columns. This is the only case in which NoDB can avoid accessing the raw data for subsequent queries.

5.1 Synopsis Construction

Our goal is to build a bi-level sampling synopsis with memory budget B out of the samples extracted from raw data used in the estimation. Remember that the samples used in estimation are driven by the accuracy ϵ. Thus, the total size can exceed memory budget B. The standard practice in online aggregation is to compute the quantities necessary for estimation – m_j, y'_j, and y''_j in our case – and then discard the sample. Since sampling from raw data is expensive, we preserve the samples in the bi-level synopsis.

We build the bi-level sample synopsis following a process similar to reservoir sampling [42]. The chunks are considered for insertion in the random order they are extracted for estimation. As long as the memory budget B is not exhausted, all the tuples extracted from a chunk are added to the synopsis—organized based on chunks. Complications appear when the memory budget is filled. In this case, we propose a *variance-driven insertion strategy*—depicted in Figure 6. The budget B is divided across the chunks in the synopsis and the new one proportionally to their local variance for the current query. The larger the internal variance, the more synopsis space a chunk gets. As shown in Figure 6, this requires dropping some of the sampled tuples inside the chunk while preserving a sample with smaller size. We realize this by discarding the tuples at the front of the random permutation based on which tuples are extracted. By following the variance-driven insertion strategy, the synopsis is guaranteed to contain a bi-level sample at any time instant. This is extremely important because processing can finish at any moment.

5.2 Synopsis Maintenance

Whenever a chunk is contained in the synopsis, we use it for estimation. If more tuples are required, we extract them starting at the end of the samples stored in the synopsis using the random permutation corresponding to the chunk. When we reach the end of the permutation, we restart from the beginning in a circular random scan (Figure 6). The new samples have to be merged with the existing ones already in the synopsis from the same chunk while satisfying the budget constraint. We have to determine the size of the merged sample and the content. The size is computed based on the same proportional variance allocation. We include in the synopsis tuples starting from the end of the new samples until we fill the allocated space. Tuples already in the synopsis are kept only if there is enough space. This strategy – depicted on the right side of Figure 6 – preserves the incremental sample generation while also refreshing the sample continuously across queries.

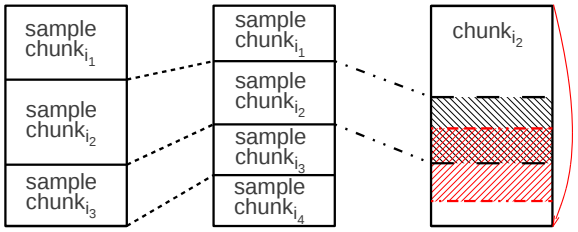

Figure 6: Maintenance of bi-level sample synopsis under new chunk insertion and existing chunk resampling.

6 EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the performance of OLA-RAW bi-level sampling across a variety of datasets – synthetic and real – and workloads—including a single query as well as a sequence of queries. Additionally, the difference among the proposed parallel bi-level sampling strategies – holistic, single-pass, and resource-aware – and with respect to chunk-level sampling is throughly analyzed. Specifically, the experiments we design are targeted to answer the following questions:

- How does OLA-RAW bi-level sampling compare with external tables and chunk-level sampling?
- How much data – chunks and tuples – are processed in order to answer a query with specified accuracy requirement?
- Is there any difference among the proposed bi-level sampling strategies? What about chunk-level sampling?
- What is the effect of the bi-level sample synopsis on the execution of a sequence of queries?

Implementation. We implement OLA-RAW based on the superscalar pipeline architecture of SCANRAW [9, 10]. Speculative loading is disabled and replaced with NoDB-style [4] caching. However, only the sample synopsis is cached in memory. In the experiments, we use CSV and FITS file formats. While SCANRAW extractors are already available, they are not immediately applicable to sampling which requires direct access to random chunks. Moreover, the tuples inside a chunk have to be accessed in random order and they have to be extracted incrementally. We introduce an estimation controller that manages the t^{eval} timing parameter and coordinates the `EXTRACT` instances. Whenever t^{eval} corresponding to an `EXTRACT` expires, the controller interrupts the instance and pulls its sample for estimation. Based on the stopping criteria and the resource utilization of the system, a decision is made on continuing the `EXTRACT` and its corresponding t^{eval} timer. The estimation

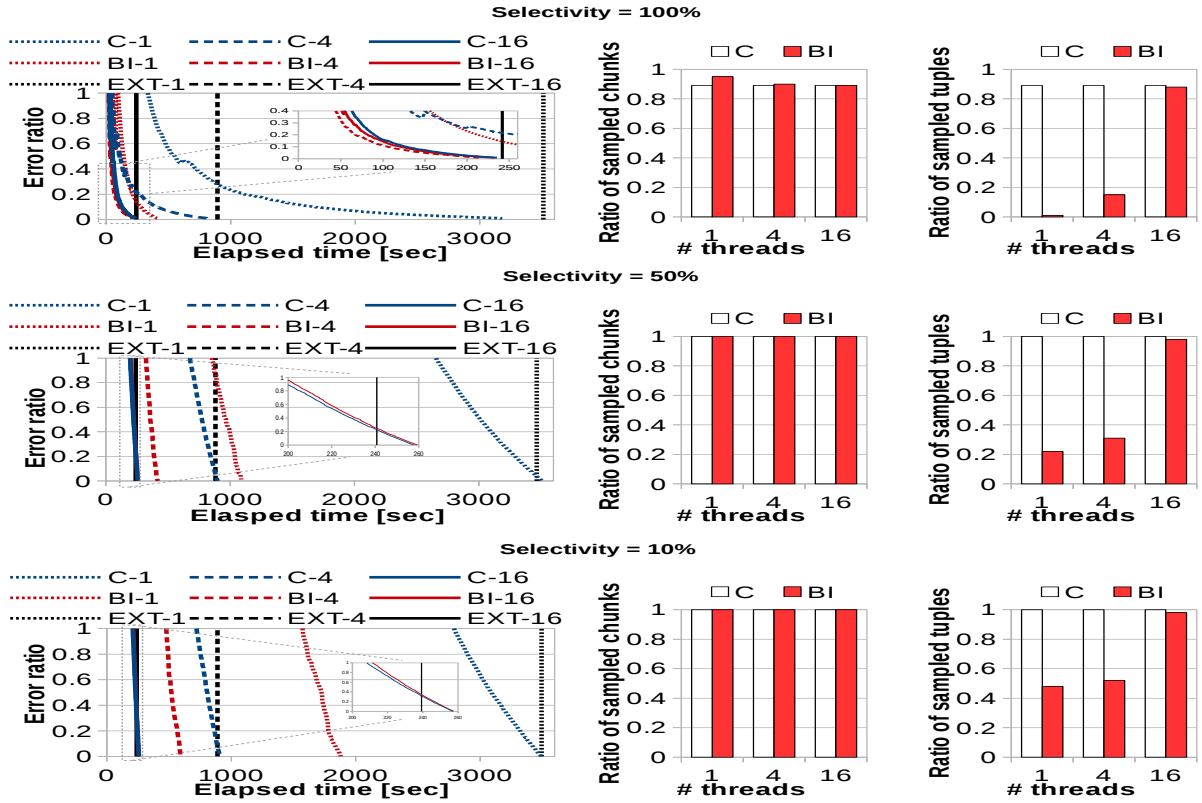

Figure 7: Results on the `ptf-csv` dataset.

controller produces an estimate and confidence bounds for the user at δ time intervals. The sample synopsis is also maintained by the controller which executes the maintenance procedure for each incoming chunk sample.

System. All experiments are executed on a standard server with 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 64 GB of memory, and four 2 TB 7200 RPM SAS hard-drives configured RAID-0 in software. Each processor has 12 MB L3 cache while each core has 128 KB L1 and 512 KB L2 local caches. The cached and buffered read rates are 3 GB/second and 565 MB/second, respectively. If the experiment consists of a single query, we always enforce data to be read from disk by cleaning the file system buffers before execution. In experiments over a sequence of queries, the buffers are cleaned only before the first query. Thus, the second and subsequent queries can access cached data. Ubuntu 14.04.2 SMP 64-bit with Linux kernel 3.13.0-43 is the operating system.

Data. We run experiments over four datasets—one synthetic and three real. Three of the datasets are in CSV format, while `ptf-fits` is the original PTF dataset in the FITS binary format. Table 2 depicts the characteristics of the datasets. `ptf-csv` and `ptf-fits` contain the same data in text and binary format, thus the difference in size. The PTF dataset contains 1 billion transient detections, i.e., tuples. Each tuple has 8 attributes, i.e., columns, 6 of which are real numbers with 10 decimal digits. `wiki` is extracted from the Wikipedia Traffic Statistics V2 dataset[4]. It contains the aggregated hits per Wikipedia page and the number of bytes transferred for each hour in March 2015. The size of this dataset is 19 GB in CSV format. The `synthetic` dataset contains randomly generated integers smaller than 1 billion grouped in tuples of 16 columns. Each column is generated using a different zipfian distribution ranging from uniform to extremely skewed. For all the datasets, the number of chunks is chosen such that the size of a chunk is in the order of tens to a hundred megabytes. This is in line with the Hadoop block size and optimizes the disk throughput.

Dataset	# Tuples	# Chunks	# Columns	Size
`ptf-csv`	1B	1000	8	68 GB
`ptf-fits`	1B	1000	8	60 GB
`wiki`	1.8B	130	4	19 GB
`synthetic`	134M	512	16	20 GB

Table 2: Datasets used in the experiments.

Methods. We perform experiments for three raw data processing methods. We use external tables (`EXT` in the figures) as a baseline for comparison. `EXT` computes the query result exactly by inspecting all the data in sequential order. There is no sampling

[4] https://aws.amazon.com/datasets/4182

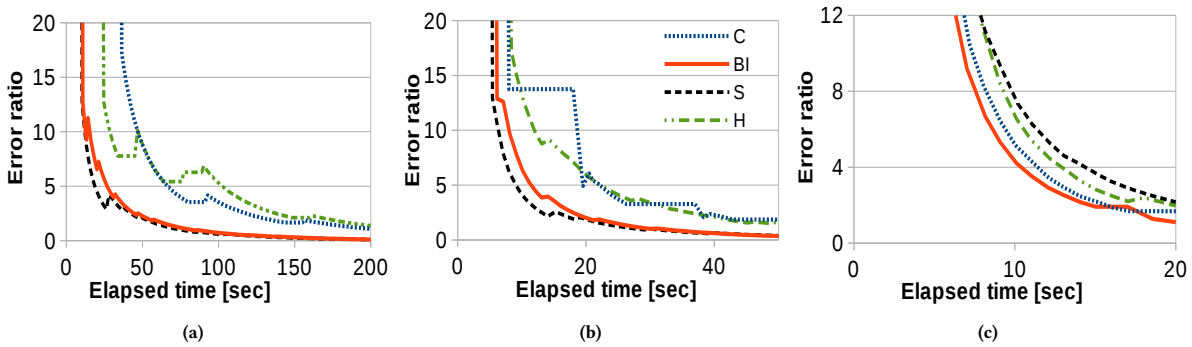

Figure 8: Sampling comparison as a function of the number of worker threads: 1 thread (a), 4 threads (b), and 16 threads (c).

or estimation, thus no overhead. The second method is parallel chunk-level sampling (C in the figures), while the third method is the proposed resource-aware OLA-RAW bi-level sampling (BI in the figures). Each method is evaluated with a different number of worker threads allocated to EXTRACT. This number is specified after the method name, e.g., BI-4 corresponds to bi-level with 4 threads.

Measurements. All our figures depict the error ratio as a function of time. The error ratio is defined as the relative confidence bounds width ($\frac{high-low}{estimate}$) measured as the ratio between the difference of the confidence interval extremes and the estimate. In all the results, the error is computed at 95% accuracy, i.e., $\epsilon = 0.95$. The estimation time interval δ is set to 1 second. This generates estimates for the user at an almost continuous rate. The number of chunks and tuples extracted in chunk-level and bi-level sampling is recorded in order to identify the difference between the two methods. These quantities are presented as the relative ratio from the entire dataset—they are always 1 for external tables.

6.1 Results

Due to space limitations, we include results only for ptf-csv and synthetic. The results for the other datasets are given in the extended version [11].

Estimation error and convergence. Figure 7 depicts a complete comparison between the three methods considered in the experiments for the ptf-csv dataset. The query used in the evaluation sums up a linear expression of the six real number attributes in the dataset. The number of tuples included in the aggregate is controlled by a selection predicate on two of the six attributes. Selectivity x% corresponds to a range predicate that covers x% of the predicate attribute domain. Since EXT computes the exact result, its error is infinite until the computation finishes, at which point it becomes zero. As the number of threads increases, the execution time for EXT decreases linearly. This shows that processing the ptf-csv dataset is CPU-bound—EXTRACT is the bottleneck. For 1 and 4 threads, the BI error is always lower than the C error at a given time instant. The reason for this is the number of tuples processed by the two methods. Although the two methods process almost the same number of chunks, it is the number of processed tuples that determines the execution time in this CPU-bound scenario. While C has to process all the tuples inside each chunk, BI stops as soon as the required accuracy is satisfied. In this case, this happens after a small number of tuples which shows that chunks contain homogeneous tuples while they are highly different among themselves. This makes sense since candidates are added to the PTF catalog based on their detection time. For 16 threads, two interesting phenomena occur. First, EXT almost catches up with the sampling methods. This is a clear sign that processing becomes I/O-bound. Second, BI reduces to C since there are enough CPUs to process all the tuples inside a chunk. As expected, low selectivity has a negative impact on estimation. Since fewer tuples are part of the estimator, it takes longer to satisfy the required accuracy. In the worst case, all the chunks and all the tuples have to be processed to achieve the required accuracy and sampling does not bring any benefit over external tables. This happens for C for all the settings when selectivity is 50% and 10%. Although BI also inspects all the chunks, it does so much faster since it does not extract all the tuples—the number of inspected tuples increases with the selectivity decrease, thus the increase in execution time. For 16 threads and low selectivity, external tables are the best alternative since sampling incurs overhead in data access.

Parallel sampling comparison. Figure 8 depicts a comparison between the parallel bi-level sampling methods discussed in the paper – holistic (H), single-pass (S), resource-aware (BI) – and chunk-level sampling (C) over the synthetic dataset. The query used in the evaluation sums up a linear expression of the 16 attributes in the dataset, without any selectivity. Since our goal is to emphasize the difference between methods, we zoom the figures on the area where the difference is more accentuated. For CPU-bound settings – 1 and 4 thread(s) – S and BI are achieving the fastest error reduction. This is because they stop as soon as the required accuracy for a chunk is reached. H and C – on the other hand – do not stop until the complete chunk is extracted. BI incurs additional delay over S due to more frequent convergence checking. Although H produces estimates earlier, it is more sensitive to fluctuations in the chunk-level estimate, especially when there is no parallelism. When multiple threads are available, the step behavior of C can be clearly detected due to chunk processing inversion. A slow chunk

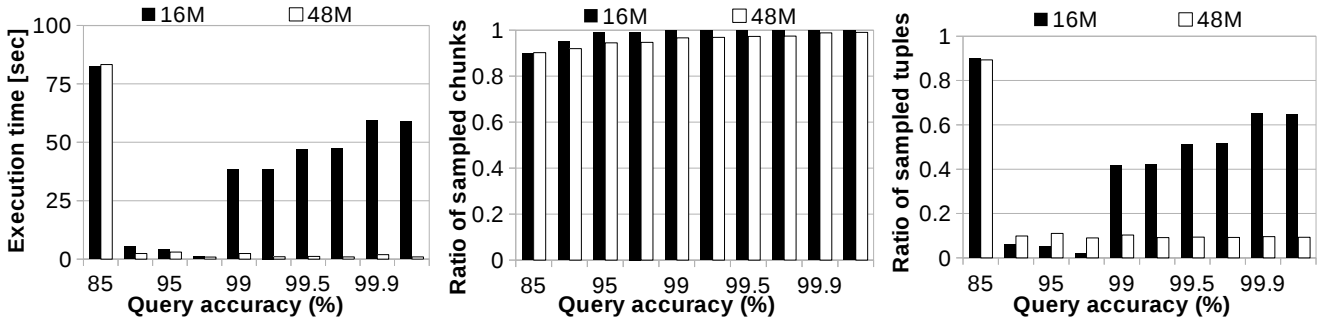

Figure 9: Sample synopsis effect on a sequence of queries with increasing accuracy. For each accuracy level, the query is executed two times.

delays estimation for all the subsequent faster chunks due to the inspection paradox. When an estimate can be finally computed, there is a steep decrease in the error. For I/O-bound settings with 16 threads, `BI` reduces to `C`, while `S` becomes the worst because it stops processing as soon as the required chunk accuracy is reached. `BI` identifies that sufficient CPU resources are available and continues to extract tuples. It also updates the timing parameter t^{eval} to reduce the estimation frequency. This is not the case for `H`, thus the larger delay. The main takeaway is that `BI` is always transforming into the best strategy, independent of the resource constraints.

Sample synopsis. The effect of the sample synopsis on the execution of a query sequence is depicted in Figure 9 for the `synthetic` dataset. We use the resource-aware bi-level sampling `BI`. We execute 10 instances of the same query at 5 increasing accuracy requirements—each query is executed twice with the same accuracy. The execution time to achieve the required accuracy, the ratio of sampled chunks, and the ratio of sampled tuples are computed for two synopsis sizes—16 MB and 48 MB. These store samples of 0.08% and 0.24% ratios from the entire dataset. The execution time for the first query is considerably larger due to accessing almost all the chunks. Since the subsequent queries can be answered based on the synopsis, their execution is much faster. In the best case, the query can be answered exclusively from the synopsis and no access to the disk is necessary. This happens for the large synopsis. The small synopsis has to expel tuples from sampled chunks in order to make space for new chunks. As a result, as the accuracy requirement increases, the same chunk has to be accessed multiple times from disk. This results in a significant increase in the execution time.

6.2 Discussion

The experimental results confirm that OLA-RAW bi-level sampling outperforms external tables and chunk-level sampling. In the best case, OLA-RAW achieves the required accuracy in as little as 10% of the time required by external tables to answer the query exactly and chunk-level sampling to achieve the same accuracy. The worst scenario for OLA-RAW is a low selectivity query in an I/O-bound context with a sufficiently large number of threads. The results also prove that a sample synopsis of less than 1% in size can provide a reduction of more than 10X in execution time across a sequence of correlated queries.

7 RELATED WORK

Raw data processing. At a high level, we can group raw data processing into two categories. In the first category, we have extensions to traditional database systems that allow raw file processing inside the execution engine. Examples include external tables [31] and various optimizations that eliminate the requirement for scanning the entire file to answer the query [4, 22, 23]. Modern database engines – e.g., Oracle, MySQL, Impala – provide external tables as a feature to directly query flat files using SQL without paying the upfront cost of loading the data into the system. NoDB [4] enhances external tables by extracting only the attributes required in the query and caching them in memory for use in subsequent queries. Data vaults [23] and SDS/Q [6] apply the same idea of query-driven just-in-time caching to scientific repositories. While the proposed sample synopsis inherits in-memory caching, it caches samples rather than full columns. Adaptive partial loading [22] materializes the cached data in NoDB to secondary storage before query execution starts—loading is query-driven. SCANRAW [9, 10] is a super-scalar adaptive external tables implementation that materializes data only when I/O resources are available. Instant loading [36] introduces vectorized SIMD implementations for tokenizing. RAW [30] and its extensions VIDa [28, 29] generate `EXTRACT` operators just-in-time for the underlying data file and the incoming query. The second category is organized around Hadoop MapReduce which processes natively raw data by including the `EXTRACT` code in the Map and Reduce functions. Invisible loading [1] focuses on eliminating the `EXTRACT` code by loading the already converted data inside a database. While similar to adaptive partial loading, instead of saving all the tuples into the database, only a fraction of tuples are stored for every query. None of these solutions supports sampling over raw data and estimation—the central contribution of this work.

Online aggregation. The database online aggregation literature has its origins in the seminal paper by Hellerstein et al. [21]. We can broadly categorize this body of work into system design [5, 16, 25, 40], online join algorithms [8, 19, 26, 34], online algorithms for estimations other than join [24, 43, 44], and methods to derive confidence bounds [18]. The distributed online aggregation literature is not as rich. Luo et al. [35] extend the ripple join algorithm [19] to a distributed setting. Wu et al. [45] 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. In subsequent work, Wu et al. [46] tackle online aggregation over multiple queries. The third piece of relevant work is online aggregation in MapReduce (Hadoop or Spark). BlinkDB [2, 3] implements a multi-stage approximation mechanism based on pre-computed sampling synopses of multiple sizes, while EARL [32] and ABS [48] use bootstrapping to produce multiple estimators from the same sample. iOLAP [47] models online aggregation as incremental view maintenance with uncertainty propagation. Sample+Seek [15] introduces measure-biased sampling to provide error guarantees for `GROUP BY` queries with many groups. Quickr [27] injects single-pass samplers in query execution plans to generate one-time approximate results. With almost no exceptions, all of this work is based on tuple-based sampling. The inadequacy of this type of sampling for database processing has been recognized in [7] where cardinality estimators based on chunk-level sampling are introduced. This type of sampling is later applied to parallel online aggregation in Hadoop [13, 38]. The only application of bi-level Bernoulli sampling to database processing is given in [20]. The last two solutions are the closest to OLA-RAW and they are discussed in detail throughout the paper. Our main contribution to online aggregation is the design of parallel bi-level sampling estimators that avoid the inspection paradox.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present OLA-RAW, a bi-level sampling scheme for parallel online aggregation over raw data. OLA-RAW sampling is query-driven and performed exclusively in-situ during query execution, without data reorganization. In order to avoid the expensive conversion cost, OLA-RAW builds and maintains incrementally a memory-resident bi-level sample synopsis. We implement OLA-RAW inside a modern in-situ data processing system and evaluate its performance across several real and synthetic datasets and file formats. Our results confirm that OLA-RAW bi-level sampling outperforms external tables and chunk-level sampling – by as much as $10X$ – and leads to a focused data exploration process that avoids unnecessary work and discards uninteresting data. In future work, we plan to perform a thorough investigation of the estimator sensitivity to the chunk size. This is a well-known problem for bi-level sampling. Adaptive solutions that change the chunk size dynamically at runtime are an interesting direction to pursue.

Acknowledgments. This work is supported by a U.S. Department of Energy Early Career Award (DOE Career).

REFERENCES

[1] A. Abouzied et al. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT/ICDT 2013*.
[2] S. Agarwal et al. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB* 5, 12 (2012).
[3] S. Agarwal et al. Knowing When You're Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *SIGMOD 2014*.
[4] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD 2012*.
[5] R. Avnur et al. CONTROL: Continuous Output and Navigation Technology with Refinement On-Line. In *SIGMOD 1998*.
[6] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *SIGMOD 2014*.
[7] S. Chaudhuri, G. Das, and U. Srivastava. Effective Use of Block-Level Sampling in Statistics Estimation. In *SIGMOD 2004*.
[8] S. Chen et al. PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. In *SIGMOD 2010*.
[9] Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD 2014*.
[10] Y. Cheng and F. Rusu. SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading. *ACM TODS* 40, 3 (2015).
[11] Y. Cheng, W. Zhao, and F. Rusu. OLA-RAW: Scalable Exploration over Raw Data. *arXiv 1702.00358*. (2017).
[12] W. Cochran. 1977. *Sampling Techniques*. Wiley.
[13] T. Condie et al. MapReduce Online. In *NSDI 2010*.
[14] G. Cormode, M. Garofalakis, P. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases* 4, 1–3 (2012).
[15] B. Ding et al. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *SIGMOD 2016*.
[16] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-Charging Estimate Convergence in DBO. *PVLDB* 2, 1 (2009).
[17] M. Garofalakis and P. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *VLDB 2001*.
[18] P. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *SSDBM 1997*.
[19] P. Haas and J. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD 1999*.
[20] P. Haas and C. König. A Bi-Level Bernoulli Scheme for Database Sampling. In *SIGMOD 2004*.
[21] J. Hellerstein, P. Haas, and H. Wang. Online Aggregation. In *SIGMOD 1997*.
[22] S. Idreos et al. Here Are My Data Files. Here Are My Queries. Where Are My Results? In *CIDR 2011*.
[23] M. Ivanova, M. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM 2012*.
[24] C. Jermaine et al. Online Estimation for Subset-Based SQL Queries. In *VLDB 2005*.
[25] C. Jermaine et al. Scalable Approximate Query Processing with the DBO Engine. In *SIGMOD 2007*.
[26] C. Jermaine et al. The Sort-Merge-Shrink Join. *ACM TODS* 31, 4 (2006).
[27] S. Kandula et al. Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters. In *SIGMOD 2016*.
[28] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries Over Heterogeneous Data Through Engine Customization. *PVLDB* 9, 12 (2016).
[29] M. Karpathiotakis et al. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *CIDR 2015*.
[30] M. Karpathiotakis et al. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014).
[31] M. Kornacker et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015*.
[32] N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB* 5, 10 (2012).
[33] N. Law et al. The Palomar Transient Factory: System Overview, Performance and First Results. *arXiv 0906.5350* (2009).
[34] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander Join: Online Aggregation via Random Walks. In *SIGMOD 2016*.
[35] G. Luo, C. Ellmann, P. Haas, and J. Naughton. A Scalable Hash Ripple Join Algorithm. In *SIGMOD 2002*.
[36] T. Muhlbauer, W. Rodiger, R. Seilbeck et al. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013).
[37] F. Olken. Random Sampling from Databases. (1993). UC Berkeley.
[38] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB* 4, 11 (2011).
[39] C. Qin and F. Rusu. PF-OLA: A High-Performance Framework for Parallel Online Aggregation. *DAPD* 32, 3 (2014).
[40] F. Rusu, F. Xu, L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The DBO Database System. In *SIGMOD 2008*.
[41] S. Thompson. 2012. *Sampling*. Wiley.
[42] J. Vitter. Random Sampling with a Reservoir. *ACM TOMS*. 11, 1 (1985).
[43] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial Online Sampling and Aggregation. *PVLDB* 9, 3 (2016).
[44] M. Wu and C. Jermaine. A Bayesian Method for Guessing the Extreme Values in a Data Set. In *VLDB 2007*.
[45] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan. Distributed Online Aggregation. *PVLDB* 2, 1 (2009).
[46] S. Wu et al. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD 2010*.
[47] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *SIGMOD 2016*.
[48] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The Analytical Bootstrap: A New Method for Fast Error Estimation in Approximate Query Processing. In *SIGMOD 2014*.
[49] W. Zhao, Y. Cheng, and F. Rusu. Vertical Partitioning for Query Processing over Raw Data. In *SSDBM 2015*.