IncApprox: A Data Analytics System for Incremental Approximate Computing

Dhanya R Krishnan
TU Dresden, Germany
dhanya.krishnan@icloud.com

Do Le Quoc
TU Dresden, Germany
do.le_quoc@tu-dresden.de

Pramod Bhatotia
TU Dresden, Germany
bhatotia@mpi-sws.org

Christof Fetzer
TU Dresden, Germany
christof.fetzer@tu-dresden.de

Rodrigo Rodrigues
IST (Univ. Lisbon) & INESC-ID
rodrigo.rodrigues@inesc-id.pt

ABSTRACT

Incremental and approximate computations are increasingly being adopted for data analytics to achieve low-latency execution and efficient utilization of computing resources. Incremental computation updates the output *incrementally* instead of re-computing everything from scratch for successive runs of a job with input changes. Approximate computation returns an *approximate* output for a job instead of the exact output.

Both paradigms rely on computing over a subset of data items instead of computing over the entire dataset, but they differ in their means for skipping parts of the computation. Incremental computing relies on the *memoization* of intermediate results of sub-computations, and reusing these memoized results across jobs. Approximate computing relies on representative *sampling* of the entire dataset to compute over a subset of data items.

In this paper, we observe that these two paradigms are complementary, and can be married together! Our idea is quite simple: *design a sampling algorithm that biases the sample selection to the memoized data items from previous runs*. To realize this idea, we designed an online stratified sampling algorithm that uses self-adjusting computation to produce an incrementally updated approximate output with bounded error. We implemented our algorithm in a data analytics system called INCAPPROX based on Apache Spark Streaming. Our evaluation using micro-benchmarks and real-world case-studies shows that INCAPPROX achieves the benefits of both incremental and approximate computing.

1. INTRODUCTION

Big data analytics systems are an integral part of modern online services. These systems are extensively used for transforming raw data into useful information. Much of this raw data arrives as a continuous data stream and in huge volumes, requiring real-time stream processing based on parallel and distributed computing frameworks [2, 6, 19, 47, 60, 72].

Near real-time processing of data streams has two desirable, but contradictory design requirements [60, 72]: *(i)* achieving low latency; and *(ii)* efficient resource utilization. For instance, the low-latency requirement can be met by employing more computing resources and parallelizing the application logic over the distributed infrastructure. Since most data analytics frameworks are based on the data-parallel programming model [34], almost linear scalability can be achieved with increased computing resources. However, low-latency comes at the cost of lower throughput and ineffective utilization of the computing resources. Moreover, in some cases, processing all data items of the input stream would require more than the available computing resources to meet the desired SLAs or the latency guarantees.

To strike a balance between these two contradictory goals, there is a surge of new computing paradigms that prefer to compute over a subset of data items instead of the entire data stream. Since computing over a subset of the input requires less time and resources, these computing paradigms can achieve bounded latency and efficient resource utilization. In particular, two such paradigms are incremental and approximate computing.

Incremental computing. Incremental computation is based on the observation that many data analytics jobs operate incrementally by repeatedly invoking the same application logic or algorithm over an input data that differs slightly from that of the previous invocation [21, 44, 47]. In such a workflow, small, localized changes to the input often require only small updates to the output, creating an opportunity to update the output incrementally instead of recomputing everything from scratch [10, 18]. Since the work done is often proportional to the change size rather than the total input size, incremental computation can achieve significant performance gains (low latency) and efficient utilization of computing resources [22, 24, 64].

The most common way for incremental computation is to rely on programmers to design and implement an application-specific incremental update mechanism (or a *dynamic algorithm*) for updating the output as the input changes [25, 29, 35, 39, 43]. While dynamic algorithms can be asymptotically more efficient than re-computing everything from scratch, research in the algorithms community shows that these algorithms can be difficult to design, implement and maintain even for simple problems. Furthermore, these algorithms are studied mostly in the context of the uniprocessor computing model, making them ill-suited for parallel and distributed settings which is commonly used for large-scale data analytics.

Recent advancements in self-adjusting computation [10, 12, 45, 46] overcome the limitations of dynamic algorithms. Self-adjusting computation transparently benefits existing applications, without requiring the design and implementation of dy-

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2883026 .

namic algorithms. At a high level, self-adjusting computation enables incremental updates by creating a dynamic dependence graph of the underlying computation, which records control and data dependencies between the sub-computations. Given a set of input changes, self-adjusting computation performs change propagation, where it reuses the *memoized* intermediate results for all sub-computations that are unaffected by the input changes, and re-computes only those parts of the computation that are transitively affected by the input change. As a result, self-adjusting computation computes only on a subset (*"delta"*) of the computation instead of re-computing everything from scratch.

Approximate computing. Approximate computation is based on the observation that many data analytics jobs are amenable to an approximate, rather than the exact output [36, 58, 61, 66]. For such an approximate workflow, it is possible to trade accuracy by computing over a partial subset instead of the entire input data to achieve low latency and efficient utilization of resources.

Over the last two decades, researchers in the database community proposed many techniques for approximate computing including sampling [15, 41], sketches [33], and online aggregation [48]. These techniques make different trade-offs with respect to the output quality, supported query interface, and workload. However, the early work in approximate computing was mainly targeted towards the centralized database architecture, and it was unclear whether these techniques could be extended in the context of big data analytics.

Recently, sampling based approaches have been successfully adopted for distributed data analytics [14, 42]. These systems show that it is possible to have a trade-off between the output accuracy and performance gains (also efficient resource utilization) by employing sampling-based approaches for computing over a *subset* of data items. However, these "big data" systems target batch processing workflow and cannot provide required low-latency guarantees for stream analytics.

The marriage. In this paper, we make the observation that the two computing paradigms, incremental and approximate computing, are complementary. Both computing paradigms rely on computing over a subset of data items instead of the entire dataset to achieve low latency and efficient cluster utilization. Therefore, we propose to combine these paradigms together in order to leverage the benefits of both. Furthermore, we achieve incremental updates without requiring the design and implementation of application-specific dynamic algorithms, and support approximate computing for stream analytics.

The high-level idea is to design a sampling algorithm that biases the sample selection to the memoized data items from previous runs. We realize this idea by designing an online sampling algorithm that selects a representative subset of data items from the input data stream. Thereafter, we bias the sample to include data items for which we already have memoized results from previous runs, while preserving the proportional allocation of data items of different (strata) distributions. Next, we run the user-specified streaming query on this biased sample by making use of self-adjusting computation and provide the user an incrementally updated approximate output with error bounds.

We implemented our algorithm in a system called INCAPPROX based on Apache Spark Streaming [5], and evaluated its effectiveness by applying INCAPPROX to various micro-benchmarks. Furthermore, we report our experience on applying INCAPPROX on two real-world case-studies: *(i)* real-time network monitoring, and *(ii)* data analytics on a Twitter stream.

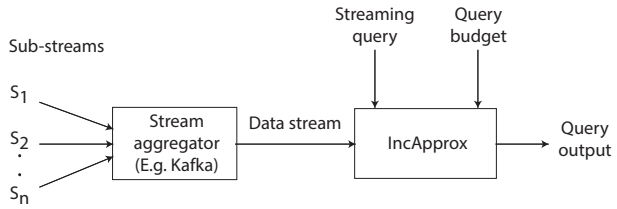

Figure 1: System overview

Our evaluation using real-world case-studies shows that INCAPPROX achieves a speedup of $\sim 2\times$ over the native Spark Streaming execution, and $\sim 1.4\times$ over the individual speedups of both incremental and approximate computing.

2. OVERVIEW

2.1 System Overview

INCAPPROX is designed for real-time data analytics on online data streams. Figure 1 depicts the high-level design of INCAPPROX. The online data stream consists of data items from diverse sources of events or sub-streams. We use a stream aggregator (such as Apache Kafka [7], Apache Flume [3], Amazon Kinesis [1], etc.) that integrates data from these sub-streams, and thereafter, the system reads this integrated data stream as the input. We facilitate user querying on this data stream by providing a user interface that consists of a streaming query and a query budget. The user submits the streaming query to the system as well as specifies a query budget. The query budget can either be in the form of latency guarantees/SLAs for data processing, desired result accuracy, or computing resources available for query processing. Our system makes sure that the computation done over the data remains within the specified budget. To achieve this, the system makes use of a mixture of incremental and approximating computing for real-time processing over the input data stream, and emits the query result along with the confidence interval or error bounds.

2.2 Design Goals

The goals of the INCAPPROX system are to:

- *Provide application transparency:* We aim to support unmodified applications for stream processing, i.e., the programmers do not have to design and implement application-specific dynamic algorithms or sampling techniques.
- *Guarantee query budget:* We aim to provide an adaptive execution interface, where the users of the system can specify their query budget in terms of tolerable latency/SLAs, desired result accuracy, or the available cluster resources, and our system guarantees the processing within the budget.
- *Improve efficiency:* We aim to achieve high efficiency with a mix of incremental and approximate computing.
- *Guarantee a confidence level:* We aim to provide a confidence level for the approximate output, i.e., the accuracy of the output will remain within an error range.

2.3 System Model

Before we explain the design of INCAPPROX, we present the system model assumed in this work.

Programming model. Our system supports a *batched streaming processing* programming model. In batched stream processing, the online data stream is divided into small batches or sets of

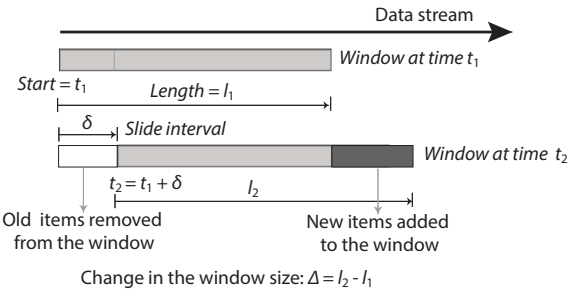

Figure 2: Sliding window computation over data stream

records; and for each batch a distributed data-parallel job is launched to produce the output.

As opposed to the trigger-based programming model (see [72] for details), the batched streaming model provides three main advantages: *(i)* it provides simple fault tolerance based on re-computation of tasks, and efficient handling of stragglers using speculative execution; *(ii)* it provides consistent "exact-once" semantics for records processing instead of weaker semantics such as "at least once" or "at most once"; and finally, *(iii)* it provides a unified data-parallel programming model that could be utilized for batch as well as stream processing workflows. Given these advantages, the batched streaming model is widely adopted by many stream processing frameworks including Spark Streaming [5], Flink [2], Slider [19, 20], TimeStream [65], Trident [8], MapReduce Online [32], Comet [47], and NOVA [26].

Computation model. Our computation model for stream processing is *sliding window computations*. In this model (see Figure 2), the computation window slides over the input data stream, where the newly arriving input data items are added to the window and the old data items are dropped from the window as they become less relevant to the analysis.

In sliding window computations, there is a substantial overlap of data items between the two successive computation windows, especially, when the size of the window is large relative to the slide interval. This overlap of unchanged data-items provides an opportunity to update the output incrementally.

Assumptions. Our system makes the following assumptions. We discuss these assumptions and the different possible methods to enforce them in § 6.

1. We assume that the input stream is stratified based on the source of event, i.e., the data items within each stratum follow the same distribution, and are mutually independent. Here a *stratum* refers to one sub-stream. If multiple sub-streams have the same distribution, they are combined to form a stratum.
2. We assume the existence of a virtual function that takes the user specified budget as the input and outputs the sample size for each window based on the budget.
3. We assume that the memoized results for incremental computation are stored in the way that is fault-tolerant.

Lastly, we assume a time-based window length, and based on the arrival rate, the number of data items within a window may vary accordingly. Note that this assumption is consistent with the sliding window APIs in the aforementioned systems.

2.4 Building Blocks

Our system leverages several computational and statistical techniques to achieve the goals discussed in § 2.2. Next, we briefly describe these techniques and the motivation behind our design choices.

Stratified sampling. In a streaming environment, since the window size might be very large, for a realistic rate of execution, we perform approximation using samples taken within the window. But the data stream might consist of data from disparate events. As such, we must make sure that every sub-stream is considered fairly to have a representative sample from each sub-stream. For this we use stratified sampling [15]. Stratified sampling ensures that data from every stratum is selected and none of the minorities are excluded. For statistical precision, we use proportional allocation of each sub-stream to the sample [16]. It ensures that the sample size of each sub-stream is in proportion to the size of sub-stream in the whole window.

Self-adjusting computation. For incremental sliding window computations, we use self-adjusting computation [10, 12, 45, 46] to re-use the intermediate results of sub-computations across successive runs of jobs. In this technique we maintain a dependence graph between sub-computations of a job, and reuse memoized results for sub-computations that are unaffected by the changed input in the computation window.

Error estimation. For defining a confidence level on the accuracy of the approximated output, we use error estimation [31]. This specifies a confidence interval or error bound for the output, i.e., we emit the output in the following form : output $\pm$ error margin. A confidence level along with the margin of error tells how accurate is the approximate output.

3. DESIGN

In this section, we present the detailed design of INCAPPROX.

3.1 Algorithm Overview

Algorithm 1 presents an overview of our approach. The algorithm computes a user-specified streaming *query* as a sliding window computation over the input data stream. The user also specifies a query *budget* for executing the query, which is used to derive the sample size (*sampleSize*) for the window using a cost function (see § 2.3 and § 6). The cost function ensures that processing remains within the query budget.

For each window (see Figure 2), we first adjust the computation window to the current start time t by removing all old data items from the *window* (*timestamp* $< t$). Similarly, we also drop all old data items from the list of memoized items (*memo*), and the respective memoized results of all sub-computations that are dependent on those old data items.

Next, we read the new incoming data items in the *window*. Thereafter, we perform proportional stratified sampling (detailed in § 3.2) on the *window* to select a sample of size provided by the cost function. The stratified sampling algorithm ensures that samples from all strata are proportional, and no stratum is neglected.

Next, we bias the stratified sample to include items from the memoized sample, in order to enable the reuse of memoized results from previous sub-computations. The biased sampling algorithm (detailed in § 3.3) biases samples *specific to each stratum*, to ensure reuse, and at the same time, retain proportional allocation.

Thereafter, on this biased sample, we run the user specified *query* as a data-parallel job *incrementally*, i.e., we reuse the memoized results for all data items that are unchanged in the window, and update the output based on the changed (or new) data items. After the job finishes, we memoize all the items in the sample

Algorithm 1 Basic algorithm

```
User input: streaming query and query budget
Windowing parameters (see Figure 2):
t ← start time; δ ← slide interval;
begin
    window ← ∅; // List of items in the window
    memo ← ∅; // List of items memoized from the window
    sample ← ∅; // Set of items sampled from the window
    biasedSample ← ∅; // Set of items in biased sample
    foreach window in the incoming data stream do
        // Remove all old items from window and memo
        forall elements in the window and memo do
            if element.timestamp < t then
                window.remove(element);
                memo.remove(element);
            end
        end
        // Add new items to the window
        window ← window.insert(new items);
        // Cost function gives the sample size based on the budget
        sampleSize ← costFunction(budget);
        // Do stratified sampling of window (§ 3.2)
        sample ← stratifiedSampling(window, sampleSize);
        // Bias the stratified sample to include memoized items (§ 3.3)
        biasedSample ← biasSample(sample, memo);
        // Run query as an incremental data parallel job for the window (§ 3.4)
        output ← runJobIncrementally(query, biasedSample);
        // Memoize all items & respective sub-computations for sample (§ 3.4)
        memo ← memoize(biasedSample);
        // Estimate error for the output (§ 3.5)
        output±error ← estimateError(output);
        // Update the start time for the next window
        t ← t + δ;
    end
end
```

and their respective sub-computation results for reuse for the subsequent windows. The details are covered in § 3.4.

The job provides an estimated output which is bound to a range of error due to approximation. We perform error estimation (as described in § 3.5) to estimate this error bound and define a confidence interval for the result as: $output \pm error\,bound$.

The entire process repeats for the next window, with updated windowing parameters and the sample size. (Note that the query budget can be updated across windows during the course of stream processing to adapt to the user's requirements.)

3.2 Stratified Reservoir Sampling

Stratified sampling clusters the input stream into homogenous disjoint sets of strata (here *homogenous* means the items within a stratum have same distribution) and selects a random sample from each stratum. Meanwhile, reservoir sampling selects a uniform random sample of *fixed size* without replacement, from an input stream of unknown size. We perform a combined *stratified reservoir sampling*, adopted from the approach in [15], along with proportional allocation, i.e., we sample the streaming data within a sliding window by stratifying the stream, and applying reservoir sampling within each stratum proportionally. By combining these two techniques, statistical quality of the sample is maintained—as sample from *every stratum* is selected *proportionally*, and a random sample of *fixed size*—given by cost function is selected from the window.

The stratified reservoir sampling algorithm (described in Algorithm 2) uses a fixed size reservoir with size equal to the sample size. It allocates the space in the reservoir proportionally to the

Algorithm 2 Stratified reservoir sampling algorithm

```
Require: T ← Interval for re-calculation of sub-reservoir size
stratifiedSampling(window, sampleSize)
begin
    S ← ∅ // Ordered set of all strata seen so far in window
    forall item belonging to stratum S_i in window do
        S.add(S_i); // Add new stratum seen to S
        i ← Index of stratum S_i ;
        // Fill reservoir until sampleSize is reached
        if (∑_{h=1}^{|S|} |sample[h]|) < sampleSize then
            sample[i].add(item); // Add item to its sub-reservoir
        end
        else
            if T interval is passed then
                forall S_i in S do
                    i ← Index of stratum S_i ;
                    // Compute new sub-reservoir size using Equation 1
                    newSize[i] ← sample[i].computeSize();
                    if newSize[i] ≠ |sample[i]| then
                        c ← newSize[i] − |sample[i]|;
                        // Do Adaptive Reservoir Sampling
                        sample[i] ← ARS(c, sample[i], S_i);
                    end
                    else
                        // Do Conventional Reservoir Sampling
                        sample[i] ← CRS(item, sample[i], S_i);
                    end
                    // Skip items in window, if seen by ARS or CRS
                    skipItemsSeen(); // Details omitted
                end
            end
            else
                // Until T, do Conventional Reservoir Sampling
                sample[i] ← CRS(item, sample[i], S_i);
            end
        end
    end
end
```

samples from each stratum, based on number of items seen so far in the corresponding stratum. As we move forward through the window for sampling, the arrival rate of items in each stratum may change, hence the proportional allocation must be updated. Therefore, periodically, the algorithm re-allocates the space in the reservoir to ensure proportional allocation. Thereafter, based on this re-allocation, we adapt the algorithm to use an adaptive reservoir sampling (ARS) [16] for those strata whose sub-reservoir sizes are changed, and conventional reservoir sampling (CRS) [15] for those strata whose sub-reservoir sizes are unchanged. (Let reservoir consists of a group of sub-reservoirs, each for storing sample from each stratum). ARS ensures that we periodically adjust the proportional allocation (based on the arrival rate), and CRS ensures randomness in sampling technique. Once the sub-reservoir's proportional allocation is handled using ARS, the sampling technique switches back to CRS, until the next re-allocation interval.

Algorithm 2 works as follows: For each *item* seen in a window, if the stratum of the item is newly seen, then we add it to the set of strata seen so far. Initially, we fill the reservoir of sample until it is full. Here the reservoir is a store for our stratified sample $'sample'$, and can be considered as a group of sub-reservoirs of different strata such that: $|sample| = \sum_{i=0}^{|S|-1} |sample[i]|$ where S is the ordered set of all strata seen so far in the window, and $sample[i]$ is the sub-reservoir of the sample from the i^{th} stratum.

Algorithm 3 Subroutines for the stratified sampling algorithm

```
Let incomingItems[ ] represent incoming items seen when moving forward
through window
ARS(c, sample[i], S_i)
begin
    if c > 0 then
        // Add c items to sample[i] from incoming items belonging to S_i
        ∀j ∈ {0, ..., c−1} : sample[i].add(incomingItems[S_i].get(j));
    end
    else
        // Evict random c items from sample[i]
        ∀j ∈ {0, ..., c−1} :
        // random(a, b) gives a random number between [a, b]
        sample[i].remove(random(0, |sample[i]| −1));
    end
end
CRS(item, sample[i], S_i)
begin
    p ← |sample[i]| / |S_i| ; // Probability of replacement
    // Replace a random item from sample[i] with item, using probability p
    sample[i].replace(sample[i][random(0, |sample[i]| −1)],
    item, p);
end
```

We fill the reservoir by adding each *item* to its corresponding sub-reservoir, based on the stratum to which the *item* belongs.

Once the reservoir is full, then until a pre-decided periodical time interval T to re-allocate sub-reservoir sizes, we proceed with a conventional reservoir sampling (CRS). In CRS technique, for each of the further items seen in each stratum S_i, we decide with a probability $\frac{|sample[i]|}{|S_i|}$ whether to accept or reject the item, i.e., all items in a stratum have equal probability of inclusion [15]. If the item is accepted, then we replace a randomly selected item in the corresponding sub-reservoir with the accepted item.

After T interval of time, we re-allocate the sub-reservoir sizes of each stratum, to ensure proportional allocation. This T interval determines how frequently proportional allocation is verified. Thus, T is selected based on frequency of change in the arrival rate in each stratum (since change in arrival rate changes proportional allocation), by counting the number of items of each stratum per time unit at the stream aggregator. First, after interval T, we compute the size of sub-reservoir to be allocated to each i^{th} stratum at current time t'. It is computed proportional to the total number of items seen so far in the corresponding stratum within the window, using the equation:

$$|sample[i](t')| = sampleSize * \frac{|S_i|}{k} \quad (1)$$

where *sampleSize* is the total size allocated to reservoir, $|S_i|$ is the number of items seen so far in the stratum S_i and k is the total number of items seen so far in the window.

Thereafter, if the re-allocated sub-reservoir size $|sample[i](t')|$ at current point of time t' is different from the previously adjusted sub-reservoir size (i.e., if there is any change in sub-reservoir size), we proceed with ARS—to adapt according to this change in size (described in Algorithm 3) as follows: When sub-reservoir size of S_i has increased by c, then from the incoming stream, we insert c items that belong to stratum S_i, to the corresponding sub-reservoir *sample*[i]. If the sub-reservoir size has decreased by c, we evict c number of items from the sub-reservoir. This ensures that proportional allocation is retained.

Algorithm 4 Biased sampling algorithm

```
biasSample(sample, memo)
begin
    S ← sample.getAllStrata(); // Set of all strata in sample
    foreach i^th stratum S_i in S do
        x ← memo[i].size(); // no. of items memoized from S_i
        y ← sample[i].size(); // no. of items in sample from S_i
        biasedSample[i] ← ∅; // List of items in biased sample from S_i
        if x ⩾ y then
            // Add y items from memo[i] to biasedSample[i] to enable re-use
            ∀j ∈ {0, ..., y−1} : biasedSample[i].add(memo[i].get(j));
        end
        else
            // First add x items from memo[i] to biasedSample[i]
            ∀j ∈ {0, ..., x−1} : biasedSample[i].add(memo[i].get(j));
            // Fill the remaining (y − x) items from the stratified sample
            int j = 0;
            while (biasedSample[i].size() < y) do
                biasedSample[i].add(sample[i].get(j));
                j++;
            end
        end
    end
end
```

If the re-allocated sub-reservoir size of a stratum is unchanged, we proceed with CRS for the stratum as explained before.

We perform stratified reservoir sampling until the window terminates and the resulting stratified sample consists of samples from each stratum, proportional to the size of corresponding stratum seen in the whole window.

3.3 Biased Sampling

Biased sampling. Biased sampling enables result reuse by including memoized data items in the sample, but at the same time, ensures that the proportional allocation of samples from each stratum is retained. In sliding window computations where the window slides by small intervals, there is only a small change in the input based on insertion and deletion from the window (see Figure 2). Hence, we memoize and reuse the results of sub-computations whose input is unchanged. However, if we reuse *all* memoized results from the previous window, the proportional allocation is lost, since proportions in different windows may vary due to difference in the arrival rate of sub-streams. Therefore, we select the *number of memoized items for result reuse*, based on the number of items in the sample from each stratum.

Algorithm 4 describes our biased sampling algorithm. In this algorithm, we bias the sample from each stratum separately. Note that here, "memoized items" and "sample size" are specific to each stratum. The algorithm works as follows: If the number of memoized items x is greater than or equal to the sample size y, then we create a biased sample with only y items from the memoized list, and neglect the extra memoized items. If the number of memoized items is less than the sample size, then we give priority to memoized items and create a biased sample with all memoized items first, and later we add more items to this biased sample from the stratified sample until the size of biased sample becomes equal to the size of stratified sample. This ensures proportional allocation. However, some of the memoized items in *memo* might be already in the stratified sample, and this might cause duplicates in the biased sample. Therefore, in practice, we use a data structure such as a HashSet for storing *biasedSample* to remove duplicates automatically. Finally, we get

a biased sample which includes all essential memoized items as well as stratified samples based on the arrival rate, thus ensuring both reuse and proportional allocation.

Precision and accuracy in biased sampling. We define precision and accuracy in terms of statistics. An estimated result is *precise* if similar results are obtained with repeated sampling, and it is *accurate* if the estimated result is closer to the true result (a precise result doesn't necessarily be accurate always) [54]. Our stratified sample is more precise than a random sample since it considers every stratum, and uses proportional allocation. Accuracy of a stratified sample is more if *(i)* different strata have major differences and *(ii)* within each stratum, there is homogeneity [54]. Based on our assumptions in 2.3, our stratified sample is more accurate than random sample since different stratum have different distribution, and items within each stratum follow the same distribution (homogenous).

We bias the sample from each stratum separately, thus preserving the statistics of stratified sampling, i.e., after the bias, the biased sample still consists of items from each stratum in the same proportional allocation obtained from stratified sampling. Further, even though the items selected within a stratum are biased to include memoized items which belong to the same stratum, since the items follow the same distribution, there is little difference between items within a stratum.

3.4 Run Job Incrementally

Next, we run the user-specified streaming query as an *incremental* data-parallel job on the biased sample (derived in § 3.3). For that, we make use of self-adjusting computation [10, 11].

In self-adjusting computation, the computation is divided into sub-computations, and a dynamic dependence graph is constructed to record dependencies between these sub-computations. Formally, a Dynamic Dependence Graph $DDG=(V,E)$ consists of nodes (V) representing sub-computations and edges (E) representing data and control dependencies between the sub-computations. Thereafter, a change propagation algorithm is used to update the output by propagating the input changes through the dependence graph. The change propagation algorithm identifies a set of sub-computations that directly depend on the changed data and re-executes those sub-computations. This in turn leads to re-computation of other data-dependent sub-computations. Change propagation terminates when all transitively dependent sub-computations are re-computed. For all the unaffected sub-computations, the algorithm reuses memoized results from previous runs without re-computation. Lastly, results for all re-computed (or newly computed) sub-computations are memoized for the next incremental run.

Next, we illustrate the application of self-adjusting computation to a data-parallel job based on the MapReduce model [34]. (Note that our implementation is based on Spark Streaming [5], which is a generic extended version of MapReduce.) Figure 3 shows the dependence graph built based on the data-flow graph of the MapReduce model. The data-flow graph is represented by a DAG, where *map* and *reduce* tasks represent nodes (or sub-computations) in the dependence graph, and the directed edges represent the dependencies between these tasks. For an incremental run, we launch *map* tasks for the newly added data items in the sample (M_5 and M_6), and reuse the memoized results for the *map* tasks from previous runs (M_1 to M_4). The output of the newly computed *map* tasks invalidates the dependent *reduce* tasks (R_3 and R_5). However, all *reduce* tasks that are unaffected

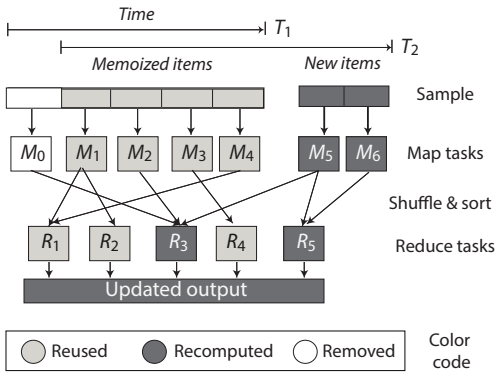

Figure 3: Run data-parallel job incrementally

by the changed input can simply reuse their memoized results without re-computation (R_1, R_2, and R_4). Lastly, we memoize the results for all *freshly* executed tasks for the next incremental run. Note that the items removed from the window also act as the input change (e.g., M_0), and sub-computations dependent on the removed items are also re-computed (e.g., R_3).

3.5 Estimation of Error Bounds

In order to provide a confidence interval for the approximate output, we estimate the error bounds due to approximation.

Approximation for aggregate functions. Aggregate functions require results based on all the data items or groups of data items in the population. But since we compute only over a small sample from the population, we get an *estimated* result based on the weightage of the sample.

Consider an input stream S, within a window, consisting of n disjoint strata S_1, S_2 ..., S_n, i.e., $S=\sum_{i=1}^{n} S_i$. Suppose the i^{th} stratum S_i has B_i items and each item j has an associated value v_{ij}. Consider an example to take sum of these values, across the whole window, represented as $\sum_{i=1}^{n}(\sum_{j=1}^{B_i} v_{ij})$. To find an approximate sum, we first select a sample from the window based on stratified and biased sampling as described in § 3, i.e., from each i^{th} stratum S_i in the window, we sample b_i items. Then we estimate the sum from this sample as: $\hat{\tau}=\sum_{i=1}^{n}(\frac{B_i}{b_i}\sum_{j=1}^{b_i} v_{ij})\pm\epsilon$ where the error bound ϵ is defined as:

$$\epsilon = t_{f,1-\frac{\alpha}{2}}\sqrt{\widehat{Var}(\hat{\tau})} \tag{2}$$

Here, $t_{f,1-\frac{\alpha}{2}}$ is the value of the t-distribution (i.e., *t-score*) with f degrees of freedom and $\alpha = 1 -$ confidence level. The degree of freedom f is expressed as:

$$f=\sum_{i=1}^{n} b_i - n \tag{3}$$

The estimated variance for sum, $\widehat{Var}(\hat{\tau})$ is represented as:

$$\widehat{Var}(\hat{\tau}) = \sum_{i=1}^{n} B_i * (B_i - b_i)\frac{s_i^2}{b_i} \tag{4}$$

where s_i^2 is the population variance in the i^{th} stratum. Since the bias sampling is such that the statistics of stratified sampling is preserved, we use the statistical theories [67] for stratified sampling to compute the error bound.

Currently, we support error estimation only for aggregate queries. For supporting queries that compute extreme values,

such as minimum and maximum, we can make use of extreme value theory [31, 52] to compute the error bounds.

Error bound estimation. For error bound estimation, we first identify the sample statistic used to estimate a population parameter, e.g., *sum*, and we select a desired confidence level, e.g., 95%. In order to compute the margin of error ϵ using t-score as given in Equation 2, the sampling distribution must be nearly normal. The Central Limit Theorem (CLT) states that when the size of sample is sufficiently large ($>= 30$), then the sampling distribution of a statistic approximates to *normal distribution*, regardless of the underlying distribution of values in the data [67]. Hence, we compute t-score using a t-distribution calculator [57], with the given degree of freedom f (see Equation 3), and cumulative probability as $1-\alpha/2$ where $\alpha = 1 -$ confidence level [54]. Thereafter, we estimate the variance using the corresponding equation for the sample statistic considered (for *sum*, the Equation is 4). Finally, we use this t-score and estimated variance of the sample statistic and compute the margin of error using Equation 2.

4. IMPLEMENTATION

We implemented INCAPPROX based on the Apache Spark Streaming framework [5]. Figure 4 presents the high-level architecture of our prototype, where the shaded boxes depict the implemented modules. In this section, we first give a brief necessary background on Spark Streaming, and next, we present the design details of the implemented modules.

Background. Spark Streaming is a scalable and fault-tolerant distributed stream processing framework. It offers batched stream processing APIs (as described in § 2.3), where a streaming computation is treated as a series of batch computations on small time intervals. For each interval, the received input data stream is first stored on a cluster's memory and a distributed file system such as HDFS [4] or Tachyon [51]. Thereafter, the input data is processed using Apache Spark [71], a distributed data-parallel job processing framework similar to MapReduce [34] or Dryad [49].

Spark Streaming is built on top of Apache Spark, which uses Resilient Distributed Datasets (RDDs) [71] for distributed data-parallel computing. An RDD is an immutable and fault-tolerant collection of elements (objects) that is distributed or partitioned across a set of nodes in a cluster. Spark Streaming extends the RDD abstraction by introducing the DStreams APIs [72], which is a sequence of RDDs arrived during a time window.

INCAPPROX implementation. Our implementation builds on the Spark Streaming APIs to implement the approximate and incremental computing mechanisms. At a high-level (see Figure 4), the input data stream is split into batches based on a pre-defined interval (e.g., one second). Each batch is defined as a sequence of RDDs. Next, the RDDs in each batch are sampled by the sampling module, with an initial sampling rate computed from the query budget using the virtual cost function. The sampled RDDs are inputs for the incremental computation module. In this module, the sampled RDDs are processed incrementally to provide the query result to the user. Finally, the error is estimated by the error estimation module. If the value of the error is higher than the error bound target, a feedback mechanism is activated to tune the sampling rate in the sampling module to provide higher accuracy in the subsequent query results. We next explain the details for the implemented modules.

I: Sampling module. The sampling module implements the approximation mechanism as described in § 3. For that, we adapt

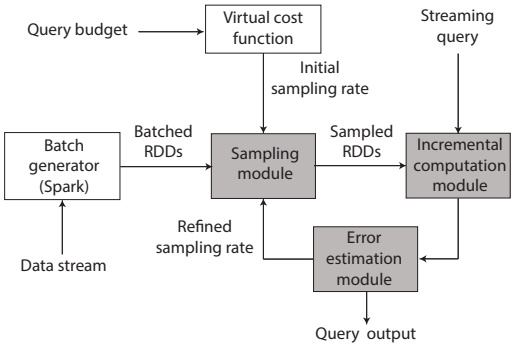

Figure 4: Architecture of INCAPPROX prototype (shaded boxes depict the implemented modules)

sampling methods available in Spark, namely *sample()*, to implement our sampling algorithm.

II: Incremental computation module. The incremental computation module implements the self-adjusting computation mechanism as described in § 3.4. For that, we reuse the caching mechanism available in Spark to memoize the intermediate results for the tasks. For the reduction operations, we adapt a windowing operation in Spark Streaming, namely *reduceByKeyAndWindow()* to incrementally update the output. Finally, the dependence graph is maintained at Spark's job controller.

III: Error estimation module. Finally, the error estimation module calculates the error bounds for the output and sends feedback to the sampling module to tune the sample size in order to satisfy the accuracy constraint. We implement the algorithm described in § 3.5 using the *Apache Common Math* library [57].

In general, our modifications in Spark Streaming are fairly straightforward, and could easily be adapted to other batched streaming processing frameworks (described in § 2.3). More importantly, we support unmodified applications since we did not modify the application programming interface.

5. EVALUATION

In this section, we first present a micro-benchmarks based evaluation (§ 5.1), and next, we report our experience on deploying INCAPPROX for the real-world case-studies (§ 5.2).

5.1 Micro-benchmarks

For analyzing the effectiveness of memoization in improving the result reuse rate, we evaluate INCAPPROX using a simulated data stream. In particular, our evaluation analyzes the impact of varying four different parameters, namely, sample size, slide interval, window size, and arrival rate for sub-streams.

We generated a synthetic data stream with three different sub-streams. Each sub-stream is generated with an independent Poisson distribution and different mean arrival rates. For the first three experiments, i.e., to analyze the impact of sample size, slide interval, and window size on memoization, we generated three sub-streams with a mean arrival rate of 3:4:5 data items per unit time respectively. To analyze the impact of the fluctuating arrival rate of events, we generated two sub-streams with fluctuating arrival rates, and kept the third sub-stream with a constant arrival rate, for a comparative analysis.

5.1.1 *What is the effect of varying sample sizes?*

We first study the effect of varying sample sizes on memoization by applying our algorithm to the synthetic data stream. For

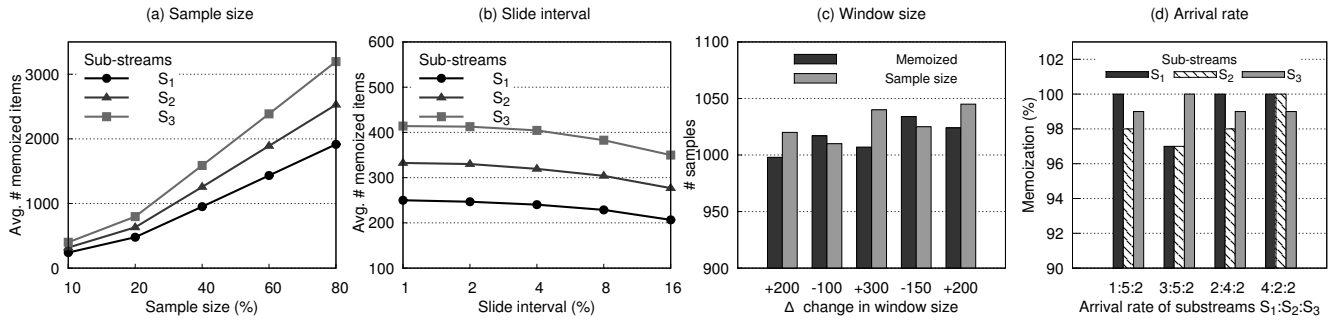

Figure 5: Effect of various parameters such as sample size, slide interval, window size and arrival rate on memoization

the experiment, we keep the other parameters—window size and slide interval—fixed. We measure the average number of memoized items from each sub-stream S_1, S_2, S_3 with different arrival rates 3:4:5 respectively, by varying the total sample size.

Figure 5 (a) shows our measurements with a fixed window size of 10,000 items, 4% slide interval (i.e., 400) and varying sample sizes (on x-axis): 10%, 20%, 40%, 60% and 80% of the window size. We observe that the average number of data items memoized is directly proportional to the sample size and the arrival rate. When the sample size increases, the average number of data items memoized increases constantly because more items from the previous window is available for memoization. We also observe a higher memoization rate for sub-streams with higher arrival rates due to proportional allocation of sub-sample sizes.

5.1.2 *What is the effect of varying slide intervals?*

Next, we evaluate the impact of varying slide intervals on memoization with constant window and sample sizes. We measure the average number of items memoized from each sub-stream with different slide intervals.

Figure 5 (b) shows our measurements with a fixed window size of 10,000 and sample size of 10% window size (i.e., 1000), but varying slide intervals (on x-axis): 1%, 2%, 4%, 8%, and 16% of the window size. We observe that when the slide interval is 1%, our algorithm memoizes an average of 99.5% of total samples, which greatly improves the reuse rate, and thus, leads to higher efficiency. As evident from the plot, when the slide interval increases, the percentage of memoized items decreases, because larger slides allow fewer samples to reuse from the previous window. We also repeated the experiments with different window sizes, but observed very similar results. Thus, the results illustrate that smaller slides (which is the usual case for an incremental workflow) allow higher memoization and thus higher result reuse.

5.1.3 *What is the effect of varying window sizes?*

Next, we evaluate the impact of varying window sizes on memoization. We measure the number of items in a sample and the number of items memoized from the previous window, and analyze the reuse rate based on this measurement. We begin our experiment with a window of 10,000 items, and then increase/decrease the window size, e.g., we first increase the window size by 200, then decrease it by 100, etc.

Figure 5 (c) shows our measurements, with a fixed 2% slide interval and 10% sample size for each corresponding window size. The x-axis represents Δ, i.e., the change in window size between two adjacent windows (see Figure 2). The figure illustrates that whenever the window size decreases (*i.e.,* Δ *is negative*), memoized samples are more than the samples needed in the current window. For example, when Δ is −100, sample size is 1010 and we have 1017 memoized items from the previous window i.e., decreasing window size can allow a 100% re-use rate, provided the slide interval is considerably low (here 2%). The figure also depicts that when window size increases (*i.e,* Δ *is positive*), the sample size is higher than the number of memoized items from the previous window, and the larger the increase in the window size, the larger is the difference between samples needed and memoized. This implies a lesser result reuse rate.

5.1.4 *What is the effect of fluctuating arrival rates?*

Lastly, we evaluate the effect of fluctuating arrival rate of sub-streams. As mentioned earlier, we generated two sub-streams, each with fluctuating arrival rates, and a third sub-stream with a constant arrival rate for the analysis. We measure the percentage of items memoized from each sub-stream.

Figure 5 (d) depicts the memoization based on fluctuating arrival rates, for a fixed window of 10,000 items and sample size of 10%. The x-axis shows the arrival rate for the three sub-streams S_1, S_2, and S_3. The figure illustrates that memoization is inversely proportional to the arrival rate. For example, for sub-stream S_1, when the arrival rate increases from 1 to 3, the percentage of memoization decreases, because the sample size gets higher due to proportional allocation, but memoized items available are lesser. When S_1's arrival rate decreases from 3 to 2, we observe that the memoization increases since we have more items memoized from the previous window. Sub-stream S_2 also depicts similar behaviour. However we notice that even though arrival rate is constant for the third sub-stream, its memoization rate differs relative to the other two sub-streams since we use a proportional allocation of sample sizes. The figure illustrates that in spite of the fluctuations in arrival rates, INCAPPROX has a memoization rate greater than 97%.

5.2 Case-studies

Next, we present our experience on deploying INCAPPROX for the following two real-world case-studies: *(i)* network traffic monitoring, and *(ii)* data analytics on Twitter stream.

Experimental setup. For the evaluation, we used a cluster of 24 nodes connected via Gigabit Ethernet (1000BaseT full duplex). Each node has 2 Intel Xeon E5405 CPUs (quad core), 8GB of RAM, and a SATA-2 hard disk, running Debian Linux 5.0 with kernel 2.6.26. We deployed INCAPPROX on 20 nodes and Apache Kafka [7] stream aggregator on the remaining 4 nodes (the setup is similar to Figure 1).

Measurements. We evaluated two key metrics: throughput and accuracy loss. The throughput is defined as the number of pro-

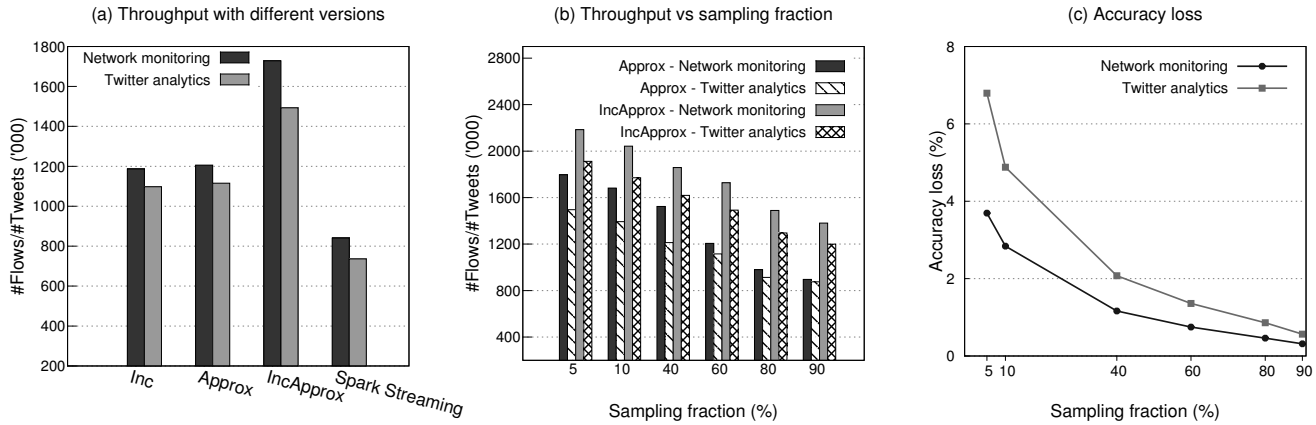

Figure 6: *(a)* Peak throughput comparison between Inc, Approx, INCAPPROX, and native Spark Streaming with the sampling fraction is set to 60%. *(b) & (c)* The effect of varying sampling fractions on throughput and accuracy with end-to-end latency 350*ms*

cessed records per second, and the accuracy loss is defined as $(approx - exact)/exact$ where *approx* and *exact* are the results obtained from approximate and native executions respectively. We report the average over 20 runs for all measurements.

Finally, to assess the individual performance benefits of incremental (Inc) and approximate (Approx) computing paradigms, we switched on Inc (incremental computation) and Approx (sampling + error estimation) modules separately. For Inc, the window size is set to 10 seconds, and the window slide interval is set to 2 seconds. For Approx, the sampling fraction is set to 60%.

5.2.1 Network Traffic Monitoring

Network traffic monitoring plays an important role in network management [50]. In this case study, we evaluated INCAPPROX's performance in a traffic monitoring application that measures the number of TCP, UDP, and ICMP packets over time.

Dataset. We used network traffic traces from CAIDA [27] captured on the high-speed Internet backbone links in Chicago (labeled as A) containing around 670 GB of data in year 2015.

Methodology. From the CAIDA traces, we created a NetFlow [30] dataset for our experiments. We developed a tool that allows us to tune the throughput of the NetFlow stream, i.e., the number of messages sent per second and the number of NetFlow records per message. The experiment was conducted for 30 minutes. The throughput of the stream is tuned to measure the system throughput. The stream starts with 1000 messages/second and continues to increase throughput until the system is exhausted. Each message from the stream contains 200 NetFlow records.

Results. Figure 6 (a) shows the throughput comparison between Approx, Inc, INCAPPROX, and native Spark Streaming. The individual throughput for approximate computing (Approx) and incremental computing (Inc) is 1.41× and 1.43× higher than native Spark Streaming execution, respectively. However, INCAPPROX performs significantly better with the combined benefits of both paradigms, an improvement of 2.1× over the native execution.

Figure 6 (b) indicates that the throughput decreases quickly with the increasing sampling fraction. With the sampling fraction of 5%, the throughput of INCAPPROX reaches up to around 2.1 million flows/second, whereas the throughput of Approx peaks at 1.8 million flows/second. At the sampling fraction of 90%, the throughput is 1.58× and 1.9× less than the throughput with 5% sampling fraction for INCAPPROX and Approx, respectively.

Figure 6 (c) shows the accuracy loss during the approximation under different sampling rates. As the sample size increases, the accuracy loss gets smaller (in other words, accuracy improves). Since we randomly sample the diverse data, the loss is not linear.

5.2.2 Twitter Analytics

Analyzing online social networks is an active research area [59]. In the second case-study, we evaluated INCAPPROX on a real-time Twitter data stream to compute trending topics.

Dataset. For this case study, we developed a crawler using the Twitter API [9] to collect publicly available tweets during three days, from September 17 to September 19, 2015.

Methodology. Since the Twitter API rate limits the number of returned tweets per request, we first dumped the crawled tweets dataset to a CSV file, and developed a tool to replay the tweets as a Twitter stream. This tool allows to control the throughput of the tweet stream. In our experiments, the throughput of the tweet stream is started with 1000 messages/second and continuously increased until the system is exhausted.

Results. Figure 6 (a) represents the throughput of each approach while processing the tweet stream. Approx and Inc achieve 1.49× and 1.51× higher throughput than native Spark Streaming. INCAPPROX is 2× better than native in terms of throughput.

Figure 6 (b) indicates that sampling fractions directly affect the throughput of INCAPPROX and Approx. With the sampling fraction of 5%, the throughput of INCAPPROX reaches up to nearly 1.9 million tweets/second, whereas this value of Approx is 1.5 million tweets/second. At the sampling fraction of 90%, the throughput is 1.6× and 1.7× less than the throughput with 5% sampling fraction for INCAPPROX and Approx, respectively.

Figure 6 (c) shows the accuracy loss with different sampling rates. The accuracy loss in case of Twitter analytics has a similar but slightly higher curve as for network monitoring.

6. DISCUSSION

The design of INCAPPROX is based on three assumptions (see § 2.3). Solving these assumptions is beyond the scope of this paper; however, in this section, we discuss some of the approaches that could be used to meet our assumptions.

I: Stratification of sub-streams. Currently we assume that sub-streams are stratified, i.e., the data items of individual sub-streams have the same distribution. However, it may not be the case.

Next, we discuss two alternative approaches, namely bootstrap [37, 38, 63] and a semi-supervised learning algorithm [56] to classify evolving data streams.

Bootstrap [37, 38, 63] is a non parametric re-sampling technique used to estimate parameters of a population. It works by randomly selecting a large number of bootstrap samples with replacement and with the same size as in the original sample. Unknown parameters of a population can be estimated by averaging these bootstrap samples. We could create such a bootstrap-based classifier from the initial reservoir of data, and the classifier could be used to classify sub-streams. Alternatively, we could employ a semi-supervised algorithm [56] to stratify a data stream. This algorithm manipulates both unlabeled and labeled data items to train a classification model.

II: Virtual cost function. Secondly, we assume that there exists a virtual function that computes the sample-size based on the user-specified query budget. The query budget could be specified as either available computing resources or latency requirements. We suggest two existing approaches—Pulsar [17] and resource prediction model [40, 55]—to design such a virtual function for given computing resources and latency requirements, respectively.

Pulsar [17] is a system that allocates resources based on tenants' demand, using a multi-resource token bucket. It provides a workload independent guarantee using a pre-advertised cost model, i.e., for each appliance and network, it advertises a virtual cost function that maps a request to its cost in tokens. We could adopt a similar cost model as follows: An "item", i.e., *a data block* to be processed, could be considered as a request and "amount of resources" needed to process it could be the cost in tokens. Since the resource budget gives total resources (*here tokens*) to be used, we could find the number of items, i.e., the sample size, that can be processed using these resources, ruling out faults and stragglers.

To find the sample-size for a given latency budget, we could use a resource prediction model based on performance metrics and QoS parameters in SLAs. Such a model could analyze the diurnal patterns of resource usage [28], e.g., off-line predictions based on pre-recorded resource usage log or predictions based on statistical machine learning [40, 55], to predict the future resource requirements based on workload and latency. From this predicted resource requirement, we could find the sample-size needed by using the above suggested method similar to Pulsar.

III: Fault tolerance. Our current algorithm does not take into account the failure of nodes in the cluster where memoized results are stored. We discuss three different approaches that could be adopted for fault tolerance if memoized results are unavailable: *(i)* we could continue processing the window without using any memoized items, albeit with lower efficiency; *(ii)* we could use a similar approach for fault tolerance as provided in Spark [71], where the lineage of memoized RDDs is used to recompute only the lost RDD partitions; *(iii)* we could make use of underlying distributed fault tolerant file-systems (HDFS [4]) to *asynchronously* replicate the memoized results.

7. RELATED WORK

INCAPPROX builds on two computing paradigms, namely, incremental and approximate computing. In this section, we survey the techniques proposed in these two paradigms.

Incremental computation. Since modifying the output of computation incrementally is asymptotically more efficient than recomputing everything from scratch, incremental computation is an active area of research for "big data" analytics. Earlier big data systems for incremental computation proposed an alternative programming model where the programmer is asked to implement an efficient incremental-update mechanism. Examples of non-transparent systems include Google's Percolator [62], and Yahoo's CBP [53]. A downside of these early proposals is that they depart from the existing programming model, and also require implementation of dynamic algorithms on per-application basis, which could be difficult to design and implement.

To overcome the limitations of the aforementioned systems, researchers proposed transparent approaches for incremental computation and job deployment [68, 69, 70]. Examples of transparent systems include Incoop [24, 23], Comet [47], DryadInc [64], Slider [19, 20], and NOVA [26]. These systems leverage the underlying data-parallel programming model such as MapReduce [34] or Dryad [49] for supporting incremental computation. Our work builds on transparent big data systems for incremental computation. In particular, we leverage the advancements in self-adjusting computation [10, 18] to improve the efficiency of incremental computation. In contrast to the existing approaches, our approach extends incremental computation with the idea of approximation, thus further improving the performance.

Approximate computation. Approximation techniques such as sampling [15, 41], sketches [33], and online aggregation [48] have been well-studied over the decades in the context of traditional (centralized) database systems. Recently proposed systems such as ApproxHadoop [42] and BlinkDB [13, 14] showed that it is possible to achieve the benefits of approximate computing also in the context of distributed big data analytics.

ApproxHadoop [42] uses multi-stage sampling [54] for approximate MapReduce [34] job execution. BlinkDB [13, 14] proposed an approximate distributed query processing engine that uses stratified sampling [15] to support ad-hoc queries with error and response time constraints. Our system builds on the advancements in approximate computing for big data analytics. However, our system is different from the existing approximate computing systems in two crucial aspects. First, unlike the existing systems, ApproxHadoop and BlinkDB, that are designed for batch processing—we target stream processing. Second, we extend approximate computing with incremental computation.

8. CONCLUSION

In this paper, we presented the marriage of incremental and approximate computations. Our approach transparently benefits unmodified applications, i.e., programmers do not have to design and implement application-specific dynamic algorithms or sampling techniques. We build on the observation that both computing paradigms rely on computing over a subset of data items instead of computing over the entire dataset. We marry these two paradigms by designing a sampling algorithm that biases the sample selection to the memoized data items from previous runs. We implemented our algorithm in a data analytics system called INCAPPROX based on Apache Spark Streaming. Our evaluation shows that INCAPPROX achieves improved benefits of low-latency execution and efficient utilization of resources.

Acknowledgements. Pramod Bhatotia is partially supported by the resilience path within CFAED at TU Dresden and an Amazon Web Services (AWS) Education Grant. Rodrigo Rodrigues is partially supported by FCT with reference UID/CEC/50021/2013 and an ERC Starting Grant (ERC-2012-StG-307732).

References

[1] Amazon Kinesis Streams. https://aws.amazon.com/kinesis/. Accessed: Jan, 2016.

[2] Apache Flink. https://flink.apache.org/. Accessed: Jan, 2016.

[3] Apache Flume. https://flume.apache.org/. Accessed: Jan, 2016.

[4] Apache Hadoop. http://hadoop.apache.org/. Accessed: Jan, 2016.

[5] Apache Spark Streaming. http://spark.apache.org/streaming. Accessed: Jan, 2016.

[6] Apache Storm. http://storm-project.net/. Accessed: Jan, 2016.

[7] Kafka - A high-throughput distributed messaging system. http://kafka.apache.org. Accessed: Jan, 2016.

[8] Trident. https://github.com/nathanmarz/storm/wiki/Trident-tutorial. Accessed: Jan, 2016.

[9] Twitter Search API. http://apiwiki.twitter.com/Twitter-API-Documentation. Accessed: Jan, 2016.

[10] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Carnegie Mellon University, 2005.

[11] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2009.

[12] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *Proceedings of the 26th Annual Symposium on Computational Geometry (SoCG)*, 2010.

[13] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2014.

[14] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[15] M. Al-Kateb and B. S. Lee. Stratified reservoir sampling over heterogeneous data streams. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Management (SSDBM)*, 2010.

[16] M. Al-Kateb, B. S. Lee, and X. S. Wang. Adaptive-size reservoir sampling over data streams. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management (SSBDM)*, 2007.

[17] S. Angel, H. Ballani, T. Karagiannis, G. O'Shea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[18] P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.

[19] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues. Slider: Incremental Sliding Window Analytics. In *Proceedings of the 15th International Middleware Conference (Middleware)*, 2014.

[20] P. Bhatotia, M. Dischinger, R. Rodrigues, and U. A. Acar. Slider: Incremental Sliding-Window Computations for Large-Scale Data Analysis. In *Technical Report: MPI-SWS-2012-004*, 2012.

[21] P. Bhatotia, P. Fonseca, U. A. Acar, B. Brandenburg, and R. Rodrigues. iThreads: A Threading Library for Parallel Incremental Computation. In *proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[22] P. Bhatotia, R. Rodrigues, and A. Verma. Shredder: GPU-Accelerated Incremental Storage and Computation. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[23] P. Bhatotia, A. Wieder, I. E. Akkus, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*, 2011.

[24] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: MapReduce for Incremental Computations. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2011.

[25] G. S. Brodal and R. Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, 2002.

[26] C. Olston et al. Nova: Continuous Pig/Hadoop Workflows. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2011.

[27] CAIDA. The CAIDA UCSD Anonymized Internet Traces 2015 (equinix-chicago-dirA). http://www.caida.org/data/passive/passive_2015_dataset.xml.

[28] R. Charles, T. Alexey, G. Gregory, H. K. Randy, and K. Michael. Towards understanding heterogeneous clouds at scale: Google trace analysis. Technical report, 2012.

[29] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 1992.

[30] B. Claise. Cisco systems NetFlow services export version 9. 2004.

[31] S. Coles. *An Introduction to Statistical Modeling of Extreme Values*. Springer, 2001.

[32] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. MapReduce Online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2010.

[33] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 2012.

[34] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2004.

[35] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications, Chapter 36: Dynamic Graphs*. 2005.

[36] A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 2000.

[37] D. M. Dziuda. *Data mining for genomics and proteomics: analysis of gene and protein expression data*. John Wiley & Sons, 2010.

[38] B. Efron and R. Tibshirani. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. *Statistical Science*, 1986.

[39] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[40] A. S. Ganapathi. Predicting and optimizing system utilization and performance via statistical machine learning. In *Technical Report No. UCB/EECS-2009-181*, 2009.

[41] M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2001.

[42] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. ApproxHadoop: Bringing Approximations to MapReduce Frameworks. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[43] L. J. Guibas. Kinetic data structures: a state of the art report. In *Proceedings of the third Workshop on the Algorithmic Foundations of Robotics (WAFR)*, 1998.

[44] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic Management of Data and Computation in Datacenters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[45] M. A. Hammer, J. Dunfield, K. Headley, N. Labich, J. S. Foster, M. Hicks, and D. Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[46] M. A. Hammer, K. Y. Phang, M. Hicks, and J. S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[47] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou. Comet: Batched Stream Processing for Data Intensive Distributed Computing. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[48] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1997.

[49] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[50] B. Li, J. Springer, G. Bebis, and M. Hadi Gunes. Review: A survey of network flow applications. *J. Netw. Comput. Appl.*, 2013.

[51] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[52] S. Liu and W. Q. Meeker. Statistical methods for estimating the minimum thickness along a pipeline. *Technometrics*, 2014.

[53] D. Logothetis, C. Olston, B. Reed, K. Web, and K. Yocum. Stateful bulk processing for incremental analytics. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2010.

[54] S. Lohr. *Sampling: Design and Analysis, 2nd Edition*. Cengage Learning, 2009.

[55] S. Mallick, G. Hains, and C. S. Deme. A resource prediction model for virtualization servers. In *Proceedings of International Conference on High Performance Computing and Simulation (HPCS)*, 2012.

[56] M. M. Masud, C. Woolam, J. Gao, L. Khan, J. Han, K. W. Hamlen, and N. C. Oza. Facing the reality of data stream classification: coping with scarcity of labeled data. *Knowledge and information systems*, 2012.

[57] C. Math. The Apache Commons Mathematics Library. `http://commons.apache.org/proper/commons-math`. Accessed: Jan, 2016.

[58] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *Proceedings of the 18th International Conference on Static Analysis (SAS)*, 2011.

[59] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2007.

[60] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[61] S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.

[62] D. Peng and F. Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[63] O. Pons. Bootstrap of means under stratified sampling. *Electronic Journal of Statistics*, 2007.

[64] L. Popa, M. Budiu, Y. Yu, and M. Isard. DryadInc: Reusing work in large-scale computations. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[65] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang. TimeStream: Reliable Stream Computation in the Cloud. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[66] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, 2011.

[67] S. K. Thompson. *Sampling*. Wiley Series in Probability and Statistics, 2012.

[68] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Brief Announcement: Modelling MapReduce for Optimal Execution in the Cloud. In *proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC)*, 2010.

[69] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Conductor: Orchestrating the Clouds. In *proceedings of the 4th international workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2010.

[70] A. Wieder, P. Bhatotia, A. Post, and R. Rodrigues. Orchestrating the Deployment of Computations in the Cloud with Conductor. In *proceedings of the 9th USENIX symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[71] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[72] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized Streams: Fault-Tolerant Streaming Computation at Scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP)*, 2013.