ApproxJoin: Approximate Distributed Joins

Do Le Quoc[†], Istemi Ekin Akkus[‡], Pramod Bhatotia[★],
Spyros Blanas[#], Ruichuan Chen[‡], Christof Fetzer[†], Thorsten Strufe[†]

[†]TU Dresden, Germany [‡]Nokia Bell Labs, Germany [★]The University of Edinburgh, UK [#]The Ohio State University, USA

ABSTRACT

A distributed join is a fundamental operation for processing massive datasets in parallel. Unfortunately, computing an equi-join over such datasets is very resource-intensive, even when done in parallel. Given this cost, the equi-join operator becomes a natural candidate for optimization using approximation techniques, which allow users to trade accuracy for latency. Finding the right approximation technique for joins, however, is a challenging task. Sampling, in particular, cannot be directly used in joins; naïvely performing a join over a sample of the dataset will not preserve statistical properties of the query result.

To address this problem, we introduce ApproxJoin. We interweave Bloom filter sketching and stratified sampling with the join computation in a new operator that preserves statistical properties of an aggregation over the join output. ApproxJoin leverages Bloom filters to avoid shuffling non-joinable data items around the network, and then applies stratified sampling to obtain a representative sample of the join output. We implemented ApproxJoin in Apache Spark, and evaluated it using microbenchmarks and real-world workloads. Our evaluation shows that ApproxJoin scales well and significantly reduces data movement, without sacrificing tight error bounds on the accuracy of the final results. ApproxJoin achieves a speedup of up to 9× over unmodified Spark-based joins with the same sampling ratio. Furthermore, the speedup is accompanied by a significant reduction in the shuffled data volume, which is up to 82× less than unmodified Spark-based joins.

KEYWORDS

Approximate join processing, multi-way joins, stratified sampling, approximate computing and distributed systems.

ACM Reference Format:
Do Le Quoc[†], Istemi Ekin Akkus[‡], Pramod Bhatotia[★], Spyros Blanas[#], Ruichuan Chen[‡], Christof Fetzer[†], Thorsten Strufe[†]. 2018. ApproxJoin: Approximate Distributed Joins. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing, Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267834

1 INTRODUCTION

The volume of digital data has grown exponentially over the last decade. A key contributor to this growth has been loosely-structured raw data that are perceived to be cost-prohibitive to clean, organize and store in a database management system (DBMS). These datasets are frequently stored in data repositories (often called "data lakes") for just-in-time querying and analytics. Extracting useful knowledge from a data lake is a challenge since it requires systems that adapt to variety in the output of different data sources and answer ad-hoc queries over vast amounts of data quickly.

To pluck the valuable information from raw data, distributed data processing frameworks such as Hadoop [2], Apache Spark [3] and Apache Flink [1] are widely used to perform ad-hoc data manipulations and then combine data from different input sources using a *join operation*. While joins are a critical building block of any analytics pipeline, they are expensive to perform, especially with regard to communication costs in distributed settings. It is not uncommon for a parallel data processing framework to take hours to process a complex join query [49].

Parallel data processing frameworks are thus embracing *approximate computing* to answer complex queries over massive datasets quickly [5, 7, 8, 34]. The approximate computing paradigm is based on the observation that approximate rather than exact results suffice if real-world applications can reason about measures of statistical uncertainty such as confidence intervals [24, 37]. Such applications sacrifice accuracy for lower latency by processing only a fraction of massive datasets. What response time and accuracy targets are acceptable for each particular problem is determined by the user who has the necessary domain expertise.

However, approximating join results by sampling is an inherently difficult problem from a correctness perspective, because uniform random samples of the join inputs cannot construct an unbiased random sample of the join output [22]. In practice, as shown in Figure 1, sampling input datasets before the join and then joining the samples sacrifices up to an order of magnitude in accuracy; sampling after the join is accurate but also $3-7\times$ slower due to unnecessary data movement to compute the complete join result.

Obtaining a *correct and precondition-free sample* of the join output in a distributed computing framework is a challenging task. Previous work has assumed some prior knowledge about the joined tables, often in the form of an offline sample or a histogram [5, 6, 8]. Continuously maintaining histograms or samples over the entire dataset (e.g., petabytes of data) is unrealistic as ad-hoc analytical queries process raw data selectively. Join approximation techniques for a DBMS, like RippleJoin [26] and WanderJoin [34], have not considered the intricacies of HDFS-based processing where random disk accesses are notoriously inefficient and data have not been

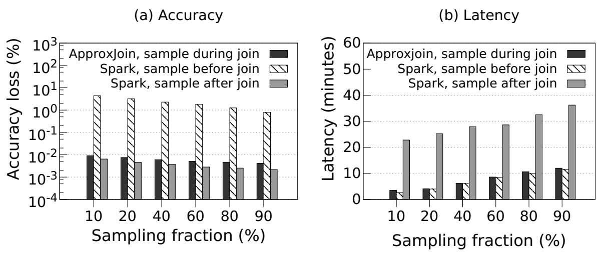

Figure 1: Comparison between different sampling strategies for distributed join with varying sampling fractions.

indexed in advance. In addition, both algorithms are designed for single-node join processing; parallelizing the optimization procedure for a data processing framework such as Apache Spark is non-trivial.

In this work, we design a novel approximate distributed join algorithm that combines a *Bloom filter* sketching technique with *stratified sampling* during the join operation and realize it in a system called ApproxJoin. As shown in Figure 1, sampling during the join produces accurate results with fast response times, such that user-defined latency and quality requirements can be met.

To achieve these goals, ApproxJoin first employs a Bloom filter to curtail redundant shuffling of tuples that will not participate in the subsequent join operations, thus reducing communication and processing overheads. This step in ApproxJoin is general and directly supports *multi-way joins*, without having to process datasets in groups of two and chain their outputs, thus, not introducing any undesired latency in the multi-way join operations. Afterwards, ApproxJoin automatically selects and progressively refines the sampling rate by estimating the cardinality of the join output using the Bloom filter. Once the sampling rate is determined, ApproxJoin performs stratified sampling over the remaining tuples to produce an answer that approximates the result of an aggregation over the complete join result. ApproxJoin uses the Central Limit Theorem and the Horvitz-Thompson estimator to remove any bias in the final result that may have been introduced by sampling without coordination from concurrent processes, producing a tight bound of the error for the accuracy of the approximation.

We implemented ApproxJoin in Apache Spark [3, 54] and evaluated its effectiveness via microbenchmarks, TPC-H queries, and real-world workloads. Our evaluation shows that ApproxJoin achieves a speedup of $6-9\times$ over Spark-based joins with the same sampling fraction. ApproxJoin leverages Bloom filtering to reduce the shuffled data volume during the join operation by $5-82\times$ compared to Spark-based systems. Without any sampling, our microbenchmark evaluation shows that ApproxJoin achieves a speedup of $2-10\times$ over the native Spark RDD join [54] and $1.06-3\times$ over a Spark repartition join. In addition, our evaluation with the TPC-H benchmark shows that ApproxJoin is $1.2-1.8\times$ faster than the state-of-the-art SnappyData system [47]. To summarize, our contributions are:

- A novel mechanism to perform stratified sampling over joins in parallel data processing frameworks, which preserves the statistical quality of the join output and reduces shuffled data size via a Bloom filter sketching technique that is directly applicable to multi-way joins.
- A progressive refinement procedure that automatically selects a sampling rate to meet user-defined latency and accuracy targets for approximate join computation.
- An extensive evaluation of an implementation of ApproxJoin in Apache Spark using microbenchmarks, TPC-H queries, and real-world workloads which shows that ApproxJoin outperforms native Spark-based joins and the state-of-the-art SnappyData system by a substantial margin.

2 OVERVIEW

ApproxJoin is designed to mitigate the overhead of distributed join operations in big data analytics systems, such as Apache Flink and Apache Spark. We facilitate joins on the input datasets by providing a simple user interface. The input of ApproxJoin consists of several datasets to be joined, as well as the join query and its corresponding query execution budget submitted by the user. The query budget can be in the form of expected latency guarantees, or the desired accuracy level. Our system ensures that the input data is processed within the specified query budget. To achieve this goal, ApproxJoin applies the approximate computing paradigm by processing only a subset of input data from the datasets to produce an approximate output with error bounds. At a high level, ApproxJoin makes use of a combination of sketching and sampling to select a subset of input datasets based on the user-specified query budget. Thereafter, ApproxJoin aggregates over the subset of input data.

ApproxJoin can also provide a subset of the join output without any aggregation (i.e., join result rows); however, such an output will not be meaningful in terms of estimating the approximation error. Hence, we assume that the query contains an algebraic aggregation function, such as SUM, AVG, COUNT, and STDEV.

Query interface. Consider the case where a user wants to perform an aggregation query after an equal-join on attribute A for n input datasets $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$, where $R_i (i = 1, \ldots, n)$ represents an input dataset. The user sends the query q and supplies a query budget q_{budget} to ApproxJoin. The query budget can be in the form of desired latency $d_{desired}$ or desired error bound $err_{desired}$. For instance, if the user wants to achieve a desired latency (e.g., $d_{desired} = 120$ seconds), or a desired error bound (e.g., $err_{desired} = 0.01$ with confidence level of 95%), he/she defines the query as follows:

```
SELECT SUM(R1.V + R2.V + ... + Rn.V)
FROM R1, R2, ..., Rn
WHERE R1.A = R2.A = ... = Rn.A
WITHIN 120 SECONDS
OR
ERROR 0.01 CONFIDENCE 95%
```

ApproxJoin executes the query and returns the most accurate query result within the desired latency which is 120 seconds, or returns the query result within the desired error bound ±0.01 at a 95% confidence level.

Design overview. The basic idea of ApproxJoin is to address the shortcomings of the existing join operations in big data systems by reducing the number of data items that need to be processed. Our first intuition is that we can reduce the latency and computation of a distributed join by removing redundant transfer of data items that are not going to participate in the join. Our second intuition is that the exact results of the join operation may be desired, but not necessarily critical, so that an approximate result with well-defined error bounds can also suffice for the user.

Figure 2 shows an overview of our approach. There are two stages in ApproxJoin for the execution of the user's query:

Stage #1: Filtering redundant items. In the first stage, ApproxJoin determines the data items that are going to participate in the join operation and filters the non-participating items. This filtering reduces the data transfer that needs to be performed over the network for the join operation. It also ensures that the join operation will not include 'null' results in the output that will require special handling, as in WanderJoin [34]. ApproxJoin employs a well-known data structure, *Bloom filter* [19]. Our filtering algorithm executes in parallel at each node that stores partitions of the input and handles multiple input tables *simultaneously*, making ApproxJoin suitable for multi-way joins without introducing undesired latency.

Stage #2: Approximation in distributed joins. In the second stage, ApproxJoin uses a sampling mechanism that is executed *during* the join process: we sample the input datasets while the cross product is being computed. This mechanism overcomes the limitations of the previous approaches and enables us to achieve low latency as well as preserve the quality of the output as highlighted in Figure 1. Our sampling mechanism is executed during the join operation and preserves the statistical properties of the output.

In addition, we combine our mechanism with *stratified sampling* [9], where tuples with distinct join keys are sampled independently with simple random sampling. As a result, data items with different join keys are selected fairly to represent the sample, and no join key will be overlooked. The final sample will contain all join keys — even the ones with few data items — so that the statistical properties of the sample are preserved.

More specifically, ApproxJoin executes the following steps for approximation in distributed joins:

Step #2.1: Determine sampling parameters. ApproxJoin employs a *cost function* to compute an optimal sample rate according to the corresponding query budget. This computation ensures that the query is executed within the desired latency and error bound parameters of the user.

Step #2.2: Sample and execute query. Using this sampling rate parameter, ApproxJoin samples during the join and then executes the aggregation query q using the obtained sample.

Step #2.3: Estimate error. After executing the query, ApproxJoin provides an approximate result together with a corresponding error bound in the form of $result \pm error\_bound$ to the user.

Note that our sampling parameter estimation provides an adaptive interface for selecting the sampling rate based on the user-defined accuracy and latency requirements. ApproxJoin adapts by activating a feedback mechanism to refine the sampling rate after learning the data distribution of the input datasets (shown by the dashed line in Figure 2).

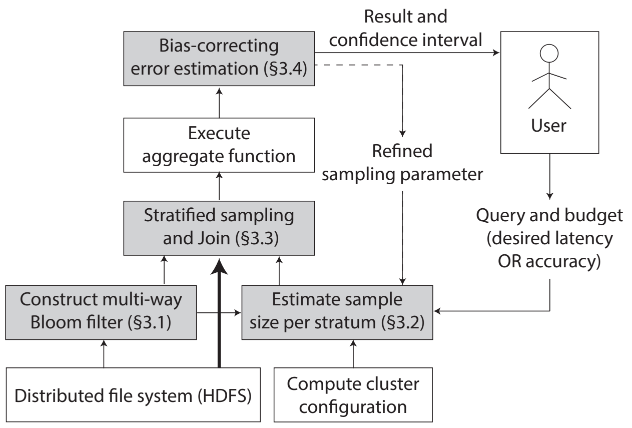
Result and confidence interval
Bias-correcting error estimation (§3.4)
User
Execute aggregate function
Refined sampling parameter
Stratified sampling and Join (§3.3)
Query and budget (desired latency OR accuracy)
Construct multi-way Bloom filter (§3.1)
Estimate sample size per stratum (§3.2)
Distributed file system (HDFS)
Compute cluster configuration

Figure 2: ApproxJoin system overview (shaded boxes depict our implemented modules in Apache Spark).

3 DESIGN

In this section, we explain the design details of ApproxJoin. We first describe how we filter redundant data items for multiple datasets to support multi-way joins (§3.1). Then, we describe how we perform approximation in distributed joins using three main steps: (1) how we determine the sampling parameter to satisfy the user-specified query budget (§3.2), (2) how our novel sampling mechanism executes during the join operation (§3.3), and finally (3) how we estimate the error for the approximation (§3.4).

3.1 Filtering Redundant Items

In a distributed setting, join operations can be expensive due to communication cost of the data items. This cost can be especially high in multi-way joins, where several datasets are involved in the join operation. One reason for this high cost is that data items not participating in the join are still shuffled through the network during the join operation.

To reduce this communication cost, we need to distinguish such redundant items and avoid transferring them over the network. In ApproxJoin, we use Bloom filters for this purpose. The basic idea is to utilize Bloom filters as a compressed set of all items present at each node and combine them to find the intersection of the datasets used in the join. This intersection will represent the set of data items that are going to participate in the join.

A Bloom filter is a data structure designed to query the presence of an element in a dataset in a rapid and memory-efficient way [19]. There are three advantages why we choose Bloom filters for our purpose. First, querying the membership of an element is efficient: it has $O(h)$ complexity, where h denotes a constant number of hash functions used in a Bloom filter. Second, the size of the filter is linearly correlated with the size of the input, but it is significantly smaller than the original input size. Lastly, constructing a Bloom filter is fast and requires only a single pass.

Bloom filters have been employed to improve distributed joins in the past [12, 33, 50, 51]. However, these proposals support only two-way joins. Although one can support joins with multiple input datasets by chaining the outputs of two-way joins, this approach would add to the latency of the join results. ApproxJoin handles multiple datasets at the same time and supports multi-way joins

Algorithm 1: Filtering using multi-way Bloom filter

```
Input:
$n$: number of input datasets
$|BF|$: size of the Bloom filter
$fp$: false positive rate of the Bloom filter
$R$: input datasets
// Build a Bloom filter for the join input datasets R
buildJoinFilter($R$, $|BF|$, $fp$)
begin
    // Build a Bloom filter for each input $R_i$
    // Executed in parallel at worker nodes
    $\forall i \in \{1...n\}$: $BF_i \leftarrow$ buildInputFilter($R_i$, $|BF|$, $fp$);
    // Merge input filters $BF_i$ for the overlap between inputs
    // Executed sequentially at master node
    $BF \leftarrow \cap_{i=1}^{n} BF_i$;
    return $BF$
// Build a Bloom filter for input $R_i$
// Executed in parallel at worker nodes
buildInputFilter($R_i$, $|BF|$, $fp$)
begin
    $|p_i|$ := number of partitions of input dataset $R_i$
    $p_i := \{p_{i,j}\}$, where $j$ = 1, ..., $|p_i|$
    //MAP PHASE
    //Initialize a filter for each partition
    forall $j$ in $\{1...|p_i|\}$ do
        p-$BF_{i,j} \leftarrow$ BloomFilter($|BF|$, $fp$);
        $\forall r_j \in p_{i,j}$: p-$BF_{i,j}$.add($r_j.key$);
    //REDUCE PHASE
    // Merge partition filters to the dataset filter
    $BF_i \leftarrow \cup_{j=1}^{|p_i|}$ p-$BF_{i,j}$;
    return $BF_i$
```

without introducing additional latency. Next, we explain in detail how our algorithm finds the intersection of multiple datasets simultaneously.

Multi-way Bloom filter. Consider the case where we want to perform a join operation between multiple input datasets R_i, where $i = 1, \cdots, n$: $R_1 \bowtie R_2 \bowtie ... \bowtie R_n$. Algorithm 1 presents the two main steps to construct the multi-way join filter. In the first step, we create a Bloom filter BF_i for each input R_i, where $i = 1, ..., n$ (lines 4-6), which is executed in parallel at all worker nodes that have the input datasets. In the second step, we combine the n dataset filters into the join filter by simply applying the logical *AND* operation between the dataset filters (lines 7-9). This operation adds virtually no additional overhead to build the join filter, because the logical *AND* operation with Bloom filters is fast, even though the number of dataset filters being combined is n instead of two.

Note that an input dataset may consist of several partitions hosted on different nodes. To build the dataset filter for these partitioned inputs, we perform a simple MapReduce job that can be executed in distributed fashion: We first build the *partition filters* p-$BF_{i,j}$, where $j = 1, \cdots, |p_i|$, and $|p_i|$ is the number of partitions for input dataset R_i during the Map phase, which is executed at the nodes that are hosting the partitions of each input (lines 15-21). Then, we combine the partition filters to obtain the dataset filter BF_i in the Reduce phase by merging the partition filters via the logical *OR* operation into the corresponding dataset filter BF_i (lines 22-24). This process is executed for each input dataset and in parallel (see `buildInputFilter()`).

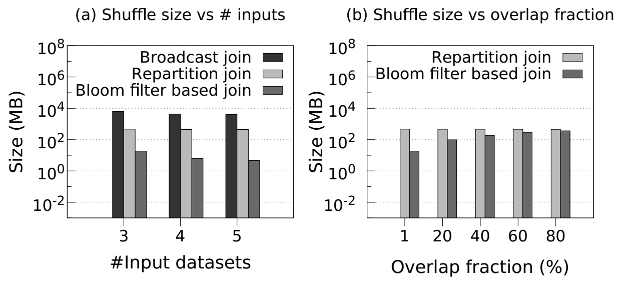

Figure 3: Shuffled size comparison between join mechanisms: (a) Varying numbers of input datasets with the overlap fraction of 1%; (b) Varying overlap fractions with three input datasets.

3.1.1 Is Filtering Sufficient? After constructing the join filter and broadcasting it to the nodes, one straightforward approach would be to complete the join operation by performing the cross product with the data items present in the intersection. Figure 3 (a) shows the advantage of performing such a join operation with multiple input datasets based on a simulation (see Appendix A.1 of the technical report [42]). With the broadcast join and repartition join mechanisms, the transferred data size gradually increases with the increasing number of input datasets. On the other hand, with the Bloom filter based join approach, the transferred data size is significantly reduced even when the number of datasets in the join operation increases.

This reduction, however, may not always be possible. Figure 3 (b) shows that even with a modest overlap fraction between three input datasets (i.e., 40%), the amount of transferred data becomes comparable with the repartition join mechanism. (In this paper, the *overlap fraction* is defined as the total number of data items participating in the join operation divided by the total number of unique data items of all inputs). Furthermore, the cross product operation will involve a significant number of data items, potentially becoming the bottleneck.

In ApproxJoin, we first filter redundant data items as aforementioned in this section. Afterwards, we check whether the overlap fraction between the datasets is small enough, such that we can meet the latency requirements of the user. If so, we perform the cross product of the data items participating in the join. In other words, we do not need approximation in this case (i.e., we compute the exact join result). If the overlap fraction is large, we continue with our approximation technique, which we describe next.

3.2 Approximation: Cost Function

ApproxJoin supports the query budget interface for users to define a desired latency ($d_{desired}$) or a desired error bound ($err_{desired}$) as described in §2. ApproxJoin ensures the join operation executed within the specified query budget by tuning the sampling parameter accordingly. In this section, we describe how ApproxJoin converts the join requirements given by a user (i.e., $d_{desired}$, $err_{desired}$) into an optimal sampling parameter. To meet the budget, ApproxJoin makes use of two types of cost functions to determine the sample size: *(i)* latency cost function, *(ii)* error bound cost function.

I: Latency cost function. In ApproxJoin, we consider the latency for the join operation being dominated by two factors: 1) the time to filter and transfer participating join data items, d_{dt}, and 2) the time to compute the cross product, d_{cp}. To execute the join operation within the delay requirements of the user, we need to estimate each contributing factor.

The latency for filtering and transferring the join data items, d_{dt}, is measured during the filtering stage (described in §3.1). We then compute the remaining allowed time to perform the join operation:

$$d_{rem} = d_{desired} - d_{dt} \tag{1}$$

To satisfy the latency requirements, the following must hold:

$$d_{cp} \leq d_{rem} \tag{2}$$

In order to estimate the latency of the cross product phase, we need to estimate how many cross products we have to perform. Imagine that the output of the filtering stage consists of data items with m distinct keys $C_1, C_2 \cdots, C_m$. To fairly select data items, we perform sampling for each join key independently (explained in §3.3). In other words, we will perform *stratified sampling*, such that each key C_i corresponds to a stratum and has B_i data items. Let b_i represent the sample size for C_i. The total number of cross products is given by:

$$CP_{total} = \sum_{1}^{m} b_i \tag{3}$$

The latency for the cross product phase would be then:

$$d_{cp} = \beta_{compute} * CP_{total} \tag{4}$$

where $\beta_{compute}$ denotes the scale factor that depends on the computation capacity of the cluster (e.g., #cores, total memory).

We determine $\beta_{compute}$ empirically via a microbenchmark by profiling the compute cluster in an offline stage. In particular, we measure the latency to perform cross products with varying input sizes. Figure 4 shows that the latency is linearly correlated with the input size, which is consistent with plenty of I/O bound queries in parallel distributed settings [8, 10, 55]. Based on this observation, we estimate the latency of the cross product phase as follows:

$$d_{cp} = \beta_{compute} * CP_{total} + \varepsilon \tag{5}$$

where ε is a noise parameter. Note that ApproxJoin computes $\beta_{compute}$ *only once* when the compute cluster is first deployed, whereas other systems perform the preprocessing steps multiple times over input data whenever it changes.

Given a desired latency $d_{desired}$, the sampling fraction $s = \frac{CP_{total}}{\sum_1^m B_i}$ can be computed as:

$$s = \left(\frac{d_{rem} - \varepsilon}{\beta_{compute}}\right) * \frac{1}{\sum_1^m B_i} = \left(\frac{d_{desired} - d_{dt} - \varepsilon}{\beta_{compute}}\right) * \frac{1}{\sum_1^m B_i} \tag{6}$$

Then, the sample size b_i of stratum C_i can be then selected as follows:

$$b_i \leq s * B_i \tag{7}$$

According to this estimation, ApproxJoin checks whether the query can be executed within the latency requirement of the user. If not, the user is informed accordingly.

II: Error bound cost function. If the user specified a requirement for the error bound, we have to execute our sampling mechanism, such that we satisfy this requirement. Our sampling mechanism

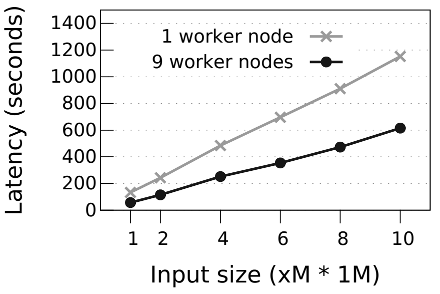

Figure 4: Latency cost function using offline profiling of the compute cluster to determine $\beta_{compute}$. The plot shows the latency of cross products with varying input sizes.

utilizes simple random sampling for each stratum (see §3.3). As a result, the error err_i can be computed as follows [35]:

$$err_i = z_{\frac{\alpha}{2}} * \frac{\sigma_i}{\sqrt{b_i}} \tag{8}$$

where b_i represents the sample size of C_i and σ_i represents the standard deviation.

Unfortunately, the standard deviation σ_i of stratum C_i cannot be determined without knowing the data distribution. To overcome this problem, we design a feedback mechanism to refine the sample size (the implementation details are in §4): For the first execution of a query, the standard deviation of σ_i of stratum C_i is computed and stored. For all subsequent executions of the query, we utilize these stored values to calculate the optimal sample size using Equation 10. Alternatively, one can estimate the standard deviation using a bootstrapping method [8, 35]. Using this method, however, would require performing offline profiling of the data.

With the knowledge of σ_i and solving for b_i gives:

$$b_i = \left(z_{\frac{\alpha}{2}} * \frac{\sigma_i}{err_i}\right)^2 \tag{9}$$

With 95% confidence level, we have $z_{\frac{\alpha}{2}} = 1.96$; thus, $b_i = 3.84 * (\frac{\sigma_i}{err_i})^2$. err_i should be less or equal to $err_{desired}$, so we have:

$$b_i \geq 3.84 * \left(\frac{\sigma_i}{err_{desired}}\right)^2 \tag{10}$$

Equation 10 allows us to calculate the optimal sample size given a desired error bound $err_{desired}$.

III: Combining latency and error bound. From Equations 7 and 10, we have a trade-off function between the latency and the error bound with confidence level of 95%:

$$d_{desired} \approx 3.84 * \left(\frac{\sigma_i}{err_{desired}}\right)^2 * \frac{\beta_{compute}}{B_i} * \left(\sum_1^m B_i\right) + d_{dt} + \varepsilon \tag{11}$$

3.3 Approximation: Sampling and Execution

In this section, we describe our sampling mechanism that executes during the cross product phase of the join operation. Executing approximation during the cross product enables ApproxJoin to have highly accurate results compared to pre-join sampling. To preserve the statistical properties of the exact join output, we combine our technique with *stratified sampling*. Stratified sampling ensures that no join key is overlooked: for each join key, we perform simple random sampling over data items independently. This method selects

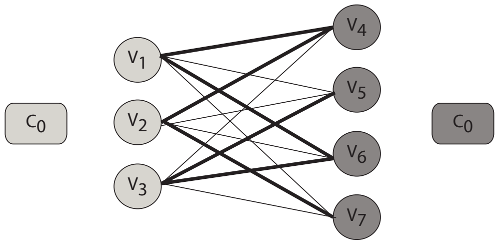

Figure 5: Cross-product the bipartite graph of join data items for key C_0. Bold lines represent sampled edges.

data items fairly from different join keys. The filtering stage (§3.1) guarantees that this selection is executed only from the data items participating in the join.

For simplicity, we first describe how we perform stratified sampling during the cross product on a single node. We then describe how the sampling can be performed on multiple nodes in parallel.

I: Single node stratified sampling. Consider an inner join example of $J = R_1 \bowtie R_2$ with a pair of keys and values, $((k_1, v_1), (k_2, v_2))$, where $(k_1, v_1) \in R_1$ and $(k_2, v_2) \in R_2$. This join operation produces an item $(k_1, (v_1, v_2)) \in J$ if and only if $k_1 = k_2$.

Consider that R_1 contains (C_0, v_1), (C_0, v_2) and (C_0, v_3), and that R_2 contains (C_0, v_4), (C_0, v_5), (C_0, v_6) and (C_0, v_7). The join operation based on key C_0 can be modeled as a complete bipartite graph (shown in Figure 5). To execute stratified sampling over the join, we perform random sampling on data items having the same join key (i.e., key C_0). As a result, this process is equal to performing *edge sampling* on the complete bipartite graph.

Sampling edges from the complete bipartite graph would require building the graph, which would correspond to computing the full cross product. To avoid this cost, we propose a mechanism to randomly select edges from the graph without building the complete graph. The function *sampleAndExecute()* in Algorithm 2 describes the algorithm to sample edges from the complete bipartite graph. To include an edge in the sample, we randomly select one endpoint vertex from each side and then yield the edge connecting these vertices (lines 19-23). To obtain a sample of size b_i, we repeat this selection b_i times (lines 17-18 and 24). This process is repeated for each join key C_i (lines 15-24).

II: Distributed stratified sampling. The sampling mechanism can naturally be adapted to execute in a distributed setting. Algorithm 2 describes how this adaptation can be achieved. In the distributed setting, the data items participating in the join are distributed to worker nodes based on the join keys using a partitioner (e.g., hash-based partitioner). A master node facilitates this distribution and directs each worker to start sampling (lines 4-5). Each worker then performs the function *sampleAndExecute()* in parallel to sample the join output and execute the query (lines 12-26).

III: Query execution. After the sampling, each node executes the input query on the sample to produce a partial query result, $result_i$, and returns it to the master node (lines 25-26). The master node collects these partial results and merges them to produce a query result (lines 6-8). The master node also performs the error bound estimation (lines 9-10), which we describe in the following subsection (§3.4) . Afterwards, the approximate query result and its error bounds are returned to the user (line 11).

Algorithm 2: : Stratified sampling over join

```
Input:
  $b_i$: sample size of join key $C_i$
  $N_{1i}$ & $N_{2i}$: set of vertices (items) in two sides of complete bipartite graph of join key $C_i$
  $m$: number of join keys
  $C$: set of all join keys (i.e., $\{\forall i \in \{1, \ldots, m\} : C_i\}$)
1  // Executed sequentially at master node
2  sampleDuringJoin()
3  begin
4      foreach $worker_i$ in $workerList$ do
5          $result_i \leftarrow worker_i$.sampleAndExecute();// Direct workers to sample and execute the query
6      $result \leftarrow \emptyset$; // Initialize empty query result
7      foreach $C_i$ in $C$ do
8          $result \leftarrow$ merge($result_i$);// Merge query results from workers
9      // Estimate error for the result
10     $result \pm error\_bound \leftarrow$ errorEstimation($result$);
11     return $result \pm error\_bound$;
12 // Executed in parallel at worker nodes
13 sampleAndExecute()
14 begin
15     foreach $C_i$ in $C$ do
16         $sample_i \leftarrow \emptyset$; // Sample of join key $C_i$
17         $count_i \leftarrow 0$;// Initialize a count to keep track # selected items
18         while $count_i < b_i$ do
19             // Select two random vertices
20             $v \leftarrow$ random($N_{1i}$);
21             $v' \leftarrow$ random($N_{2i}$);
22             // Add an edge between the selected vertices and update the sample
23             $sample_i$.add($< v, v' >$);
24             $count_i \leftarrow count_i + 1$; // Update counting
25         $result_i \leftarrow$ query($sample_i$); // Execute query over sample
26         return $result_i$;
```

3.4 Approximation: Error Estimation

As the final step, ApproxJoin computes an error bound for the approximate result. The approximate result is then provided to the user as *approxresult ± error_bound*. Our sampling algorithm (i.e., *sampleAndExecute()* in Algorithm 2) described in the previous section can produce an output with duplicate edges. For such cases, we use the Central Limit Theorem to estimate the error bounds for the output. This error estimation is possible because the sampling mechanism works as a random sampling with replacement.

It is also possible to remove the duplicate edges during the sampling process by using a hash table, and repeat the algorithm steps until we reach the desired number of data items in the sample. This approach might worsen the randomness of the sampling mechanism and could introduce bias into the sample data. In this case, we use the Horvitz-Thompson [28] estimator to remove this bias. We next explain the details of these two error estimation mechanisms.

I: Error estimation using the Central Limit Theorem. Suppose we want to compute the approximate sum of data items after the join operation. The output of the join operation contains data items with m different keys $C_1, C_2, \cdots, C_m$, each key (stratum) C_i has B_i data items and each such data item j has an associated value $v_{i,j}$. To compute the approximate sum of the join output, we sample b_i items from each join key C_i according to the parameter we computed (described in §3.2). Afterwards, we estimate the sum from this sample as $\hat{\tau} = \sum_{i=1}^{m} (\frac{B_i}{b_i} \sum_{j=1}^{b_i} v_{ij}) \pm \epsilon$, where the error bound ϵ is defined as:

$$\epsilon = t_{f,1-\frac{\alpha}{2}} \sqrt{\widehat{Var}(\hat{\tau})} \tag{12}$$

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

$$f = \sum_{i=1}^{m} b_i - m \tag{13}$$

The estimated variance for the sum, $\widehat{Var}(\hat{\tau})$, can be expressed as:

$$\widehat{Var}(\hat{\tau}) = \sum_{i=1}^{m} B_i * (B_i - b_i) \frac{r_i^2}{b_i} \tag{14}$$

Here, r_i^2 is the population variance in the i-th stratum. We use the statistical theories for stratified sampling [48] to compute the error bound.

II: Error estimation using the Horvitz-Thompson estimator. Consider the second case, where we remove the duplicate edges and resample the endpoint nodes until another edge is yielded. The bias introduced by this process can be estimated using the Horvitz-Thomson estimator. Horvitz-Thompson is an unbiased estimator for the population sum and mean, regardless of whether sampling is with or without replacement.

Let π_i be a positive number representing the probability that data item having key C_i is included in the sample under a given sampling scheme. Let y_i be the sample sum of items having key C_i: $y_i = \sum_{j=1}^{b_i} v_{ij}$. The Horvitz-Thompson estimation of the total is then computed as [48]:

$$\hat{\tau_\pi} = \sum_{i=1}^{m} (\frac{y_i}{\pi_i}) \pm \epsilon_{ht} \tag{15}$$

where the error bound ϵ_{ht} is given by:

$$\epsilon_{ht} = t_{\frac{\alpha}{2}} \sqrt{\widehat{Var}(\hat{\tau_\pi})} \tag{16}$$

where t has $n-1$ degrees freedom. The estimated variance of the Horvitz-Thompson estimation is computed as:

$$\widehat{Var}(\hat{\tau_\pi}) = \sum_{i=1}^{m} (\frac{1-\pi_i}{\pi_i^2}) * y_i^2 + \sum_{i=1}^{m} \sum_{j \neq i} (\frac{\pi_{ij} - \pi_i \pi_j}{\pi_i \pi_j}) * \frac{y_i y_j}{\pi_{ij}} \tag{17}$$

where π_{ij} is the probability that both data items having key C_i and C_j are included.

Note that the Horvitz-Thompson estimator does not depend on how many times a data item may be selected. Each distinct item of the sample is used only once [48].

4 IMPLEMENTATION

In this section, we describe the implementation details of ApproxJoin. At a high level, ApproxJoin is composed of two main modules: *(i)* filtering and *(ii)* approximation. The filtering module constructs the join filter to determine the data items participating in the join. These data items are fed to the approximation module to perform the join query within the query budget specified by the user.

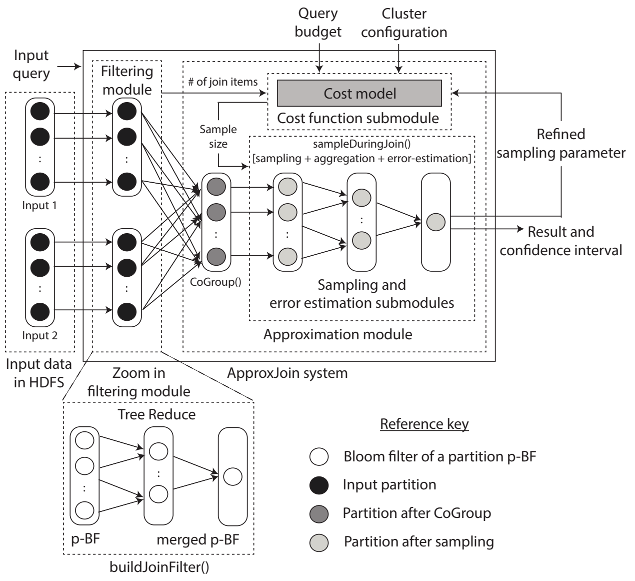

Figure 6: System implementation which shows in detail the directed acyclic graph (DAG) execution of ApproxJoin.

We implemented our design by modifying Apache Spark [3]. Spark uses Resilient Distributed Datasets (RDDs) [54] for scalable and fault-tolerant distributed data-parallel computing. An RDD is an immutable collection of objects distributed across a set of machines. To support existing programs, we provide a simple programming interface that is also based on the RDDs. In other words, all operations in ApproxJoin, including filtering and approximation, are transparent to the user. To this end, we have implemented a PairRDD for *approxjoin()* function to perform the join query within the query budget over inputs in the form of RDDs. Figure 6 shows in detail the directed acyclic graph (DAG) execution of ApproxJoin.

I: Filtering module. The join Bloom filter module implements the filtering stage described in §3.1 to eliminate the non-participating data items. A straightforward way to implement *buildJoinFilter()* in Algorithm 1 is to build Bloom filters for all partitions (p-BFs) of each input and merge them in the driver of Spark in the Reduce phase. However, in this approach, the driver quickly becomes a bottleneck when there are multiple data partitions located on many workers in the cluster. To solve this problem, we leverage the *treeReduce* scheme [14, 17, 18]. In this model, we combine the Bloom filters in a hierarchical fashion, where the reducers are arranged in a tree with the root performing the final merge (Figure 6). If the number of workers increases (i.e., ApproxJoin deployed in a bigger cluster), more layers are added to the tree to ensure that the load on the driver remains unchanged. After building the join filter, ApproxJoin broadcasts it to determine participating join items in all inputs and feed them to the approximation module.

The approximation module consists of three submodules including the cost function, sampling, and error estimation. The cost function submodule implements the mechanism in §3.2 to determine the sampling parameter according to the requirements in the query budget. The sampling submodule performs the proposed sampling mechanism (described in §3.3) and executes the join query over the filtered data items with the sampling parameter. The error

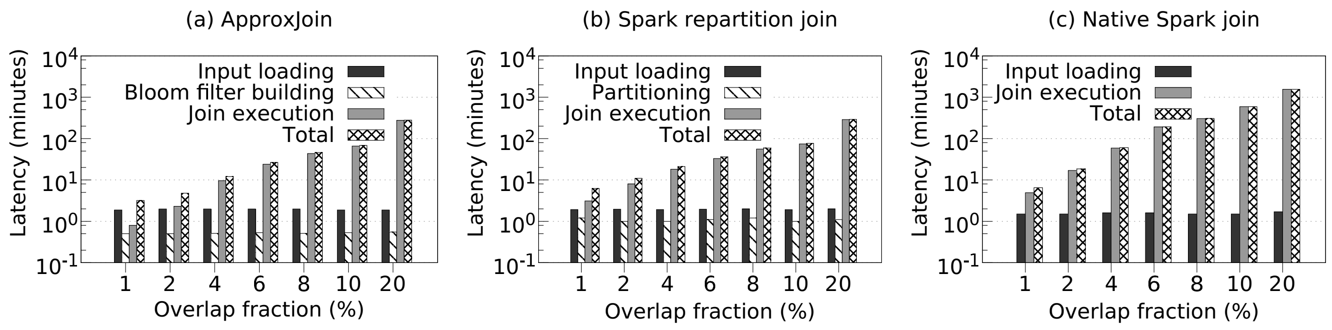

Figure 7: Benefits of filtering in two-way joins. We show the total latency and the breakdown latency of (a) ApproxJoin, (b) Spark repartition join, and (c) native Spark join.

estimation submodule computes the error-bound (i.e., confidence interval) for the query result from the sampling module (described in §3.4). This error estimation submodule also performs fine-tuning of the sample size used by the sampling submodule to meet the accuracy requirement in subsequent runs.

II: Approximation: Cost function submodule. The cost function submodule converts the query budget requirements provided by the user into the sampling parameter used in the sampling submodule. We implemented a simple cost function by building a model to convert the desired latency into the sampling parameter. To build the model, we perform offline profiling of the compute cluster. This model empirically establishes the relationship between the input size and the latency of cross product phase by computing the $\beta_{compute}$ parameter from the microbenchmarks. Afterwards, we utilize Equation 7 to compute the sample sizes.

III: Approximation: Sampling submodule. After receiving the intersection of the inputs from the filtering module and the sampling parameter from the cost function submodule, the sampling submodule performs the sampling during the join as described in §3.3. We implemented the proposed sampling mechanism in this submodule by creating a new Spark PairRDD function *sampleDuringJoin()* that executes stratified sampling during the join.

The original *join()* function in Spark uses two operations: 1) *cogroup()* shuffles the data in the cluster, and 2) *cross-product* performs the final phase in join. In our *approxjoin()* function, we replace the second operation with *sampleDuringJoin()* that implements our mechanism described in §3.3 and Algorithm 2. Note that the data shuffled by the *cogroup()* function is the output of the filtering stage. As a result, the amount of shuffled data can be significantly reduced if the overlap fraction between datasets is small. Note also that *sampleDuringJoin()* also performs the query execution as described in Algorithm 2.

IV: Approximation: Error estimation submodule. After the query execution is performed in *sampleDuringJoin()*, the error estimation submodule implements the function *errorEstimation()* to compute the error bounds of the query result. The submodule also activates a feedback mechanism to re-tune the sample sizes in the sampling submodule to achieve the specified accuracy target as described in §3.2. We use the Apache Common Math library to implement the error estimation mechanism described in §3.4.

5 EVALUATION: MICROBENCHMARKS

In this section, we present the evaluation results of ApproxJoin based on microbenchmarks and the TPC-H benchmark. In the next section, we will report evaluation based on real-world case studies.

5.1 Experimental Setup

Cluster setup. Our cluster consists of 10 nodes, each equipped with two Intel Xeon E5405 quad-core CPUs, 8GB memory and a SATA-2 hard disk, running Ubuntu 14.04.1.

Synthetic datasets. We analyze the performance of ApproxJoin using synthetic datasets following Poisson distributions with λ in the range of [10, 10000]. For the load balancing, the number of distinct join keys is set to be proportional to the number of workers.

Metrics. We evaluate ApproxJoin using three metrics: latency, shuffled data size, and accuracy loss. Specifically, the latency is defined as the total time consumed to process the join operation (including the Bloom filter building and the cross product operation); the shuffled data size is defined as the total size of the data shuffled across nodes during the join operation; the accuracy loss is defined as $(approx - exact)/exact$, where *approx* and *exact* denote the results from the executions with and without sampling, respectively.

5.2 Benefits of Filtering

The join operation in ApproxJoin consists of two main stages: *(i)* filtering stage for reducing shuffled data size, and *(ii)* sampling stage for approximate computing. In this section, we activate only the filtering stage (without the sampling stage) in ApproxJoin, and evaluate the benefits of the filtering stage.

I: Two-way joins. First, we report the evaluation results with two-way joins. Figure 7(a)(b)(c) show the latency breakdowns of ApproxJoin, Spark repartition join, and native Spark join, respectively. Unsurprisingly, the results show that building bloom filters in ApproxJoin is quite efficient (only around 42 seconds) compared with the cross-product-based join execution (around 43× longer than building bloom filters, for example, when the overlap fraction is 6%). The results also show that the cross-product-based join execution is fairly expensive across all three systems.

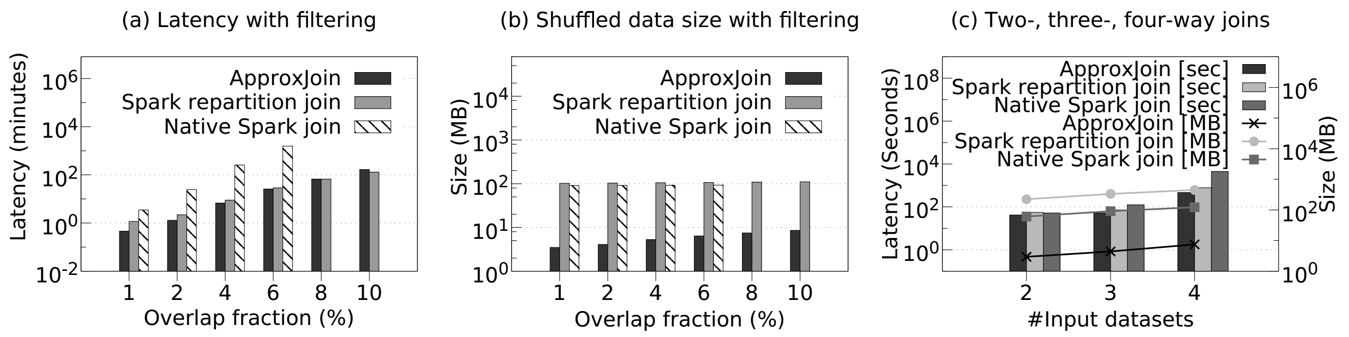

Figure 8: Benefits of filtering in multi-way joins, with different overlap fractions and different numbers of input datasets.

When the overlap fraction is less than 4%, ApproxJoin achieves 2× and 3× shorter latencies than Spark repartition join and native Spark join, respectively. However, with the increase of the overlap fraction, there is an increasingly large amount of data that has to be shuffled and the expensive cross-product operation cannot be eliminated in the filtering stage; therefore, the benefit of the filtering stage in ApproxJoin gets smaller. For example, when the overlap fraction is 10%, ApproxJoin speeds up only 1.06× and 8.2× compared with Spark repartition join and Spark native join, respectively. When the overlap fraction increases to 20%, ApproxJoin's latency does not improve (or may even perform worse) compared with the Spark repartition join. At this point, we need to activate the sampling stage of ApproxJoin to reduce the latency of the join operation, which we will evaluate in §5.3.

II: Multi-way joins. First, we present the evaluation results with multi-way joins. Specifically, we first conduct the experiment with three-way joins whereby we create three synthetic datasets with the same aforementioned Poisson distribution. We measure the latency and the shuffled data size during the join operations in ApproxJoin, Spark repartition join and native Spark join, with varying overlap fractions. Figure 8(a) shows that, with the overlap fraction of 1%, ApproxJoin is 2.6× and 8× faster than Spark repartition join and native Spark join, respectively. However, with the overlap fraction larger than 8%, ApproxJoin does not achieve much latency gain (or may even perform worse) compared with Spark repartition join. This is because, similar to the two-way joins, the increase of the overlap fraction prohibitively leads to a larger amount of data that needs to be shuffled and cross-producted. Note also that, we do not have the evaluation results for native Spark join with the overlap fractions of 8% and 10%, simply because that system runs out of memory. Figure 8(b) shows that ApproxJoin significantly reduces the shuffled data size. With the overlap fraction of 6%, ApproxJoin reduces the shuffled data size by 16.7× and 14.5× compared with Spark repartition join and native Spark join, respectively.

Next, we conduct experiments with two-way, three-way and four-way joins. In two-way joins, we use two synthetic datasets A and B that have an overlap fraction of 1%; in three-way joins, the three synthetic datasets A, B, and C have an overlap fraction of 0.33%, and the overlap fraction between any two of them is also 0.33%; in four-way joins, the four synthetic datasets have an overlap fraction of 0.25%, and the overlap fraction between any two of these datasets is also 0.25%.

Figure 8(c) presents the latency and the shuffled data size during the join operation with different numbers of input datasets. With two-way joins, ApproxJoin speeds up by 2.2× and 6.1×, and reduces the shuffled data size by 45× and 12×, compared with Spark repartition join and native Spark join, respectively. In addition, with three-way and four-way joins, ApproxJoin achieves even larger performance gain. This is because, with the increase of the number of input datasets, the number of non-join data items also increases; therefore, ApproxJoin gains more benefits from the filtering stage.

III: Scalability. Finally, we keep the overlap fraction of 1% and evaluate the scalability of ApproxJoin with different numbers of compute nodes. Figure 9(a) shows that ApproxJoin achieves a lower latency than Spark based systems. With two nodes, ApproxJoin achieves a speedup of 1.8× and 10× over Spark repartition join and native Spark join, respectively. Meanwhile, with 8 nodes, ApproxJoin achieves a speedup of 1.7× and 6× over Spark repartition join and native Spark join.

5.3 Benefits of Sampling

As shown in previous experiments, ApproxJoin does not gain much latency benefit from the filtering stage when the overlap fraction is large. To reduce the latency of the join operation in this case, we activate the second stage of ApproxJoin, i.e., the sampling stage. For a fair comparison, we re-purpose Spark's built-in sampling algorithm (i.e., stratified sampling via `sampleByKey`) to build a "sampling over join" mechanism for the Spark repartition join system. Specifically, we perform the stratified sampling over the join results after the join operation has finished in the Spark repartition join system. We then evaluate the performance of ApproxJoin, and compare it with this *extended* Spark repartition join system.

I: Latency. We measure the latency of ApproxJoin and the extended Spark repartition join with varying sampling fractions. Figure 9(b) shows that the Spark repartition join system scales poorly with a significantly higher latency as it could perform stratified sampling only after finishing the join operation.

II: Accuracy. Next, we measure the accuracy of ApproxJoin and the extended Spark repartition join. Figure 9(c) shows that the accuracy losses in both systems decrease with the increase of sampling fractions, although ApproxJoin's accuracy is slightly worse than the Spark repartition join system. Note however that, as shown in Figure 9(b), ApproxJoin achieves an order of magnitude speedup

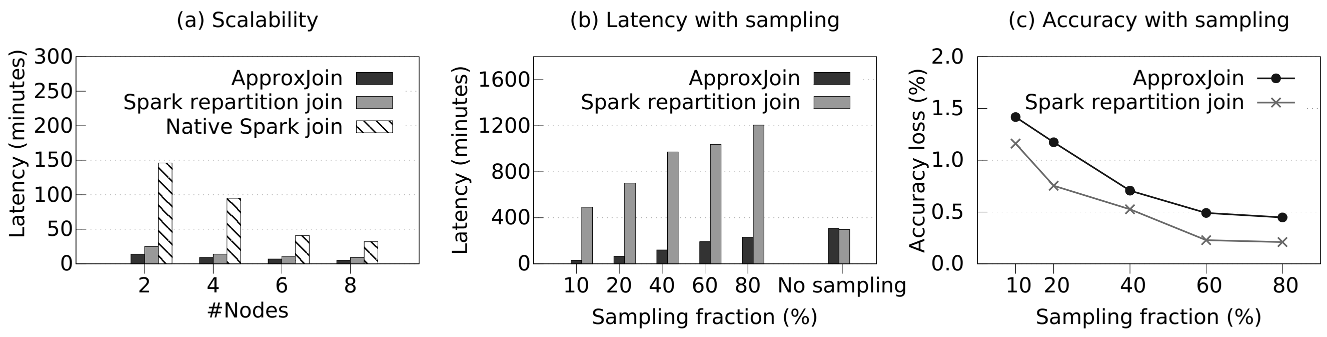

Figure 9: Comparison between ApproxJoin and Spark join systems in terms of (a) scalability, (b) latency with sampling, and (c) accuracy loss with sampling.

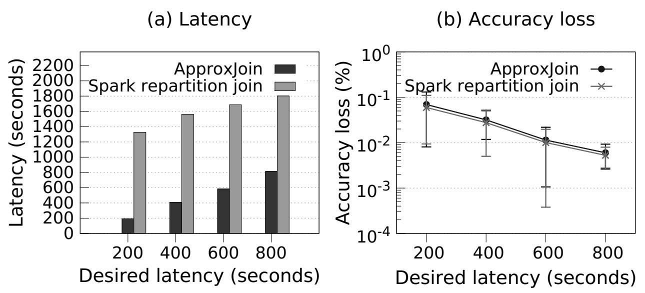

Figure 10: Effectiveness of the cost function.

compared with the Spark repartition join system since ApproxJoin performs sampling during joins.

5.4 Effectiveness of the Cost Function

ApproxJoin provides users with a query budget interface, and uses a cost function to convert the query budget into a sample size (see §3.2). In this experiment, a user sends ApproxJoin a join query along with a latency budget (i.e., the desired latency the user wants to achieve). ApproxJoin uses the cost function, whose parameter is set according to the microbenchmarks ($\beta = 4.16 * 10^{-9}$ in our cluster), to convert the desired latency to the sample size. We measure the latency of ApproxJoin and the extended Spark repartition join in performing the join operations with the identified sample size. Figure 10(a) shows that ApproxJoin can rely on the cost function to achieve the desired latency quite well (with the maximum error being less than 12 seconds). Note also that, the Spark repartition join incurs a much higher latency than ApproxJoin since it performs the sampling after the join operation has finished. In addition, Figure 10(b) shows that ApproxJoin can achieve a very similar accuracy to the Spark repartition join system.

5.5 Comparison with SnappyData using TPC-H

In this section, we evaluate ApproxJoin using TPC-H benchmark. TPC-H benchmark consists of 22 queries, and has been widely used to evaluate various database systems. We compare ApproxJoin with the state-of-the-art system — SnappyData [47].

SnappyData is a hybrid distributed data analytics framework which supports a unified programming model for transactions, OLAP and data stream analytics. It integrates GemFine, an in-memory transactional store, with Apache Spark. SnappyData inherits approximate computing techniques from BlinkDB [8] (offline sampling techniques) and the data synopses to provide interactive analytics. SnappyData does not support sampling over joins. In particular, we compare ApproxJoin with SnappyData using the TPC-H queries $Q3$, $Q4$, and $Q10$ which contain join operations. These queries are dominated by joins rather than other operations. To make a fair comparison, we only keep the join operations and remove other operations in these queries. This is to focus on the performance comparison of joins without being affected by the performance of other parts in the evaluated systems. We run the benchmark with a scale factor of 10×, i.e., 10GB datasets. If we set the scale factor even larger to 100×, the evaluated systems take many hours to process the queries and run out of memory due to the limited capacity of our cluster.

First, we use the TPC-H benchmark to analyze the performance of ApproxJoin with the filtering stage but without the sampling stage. Figure 11(a) shows the end-to-end latencies of ApproxJoin and SnappyData in processing the three queries. ApproxJoin is 1.34× faster than SnappyData in processing $Q4$ which contains only one join operation. In addition, for the query $Q3$ which consists of two join operations, ApproxJoin achieves a 1.3× speedup than SnappyData. Meanwhile, ApproxJoin speeds up by 1.2× compared with SnappyData for the query $Q10$.

Next, we evaluate ApproxJoin with both filtering and sampling stages activated. In this experiment, we perform a query to answer the question *"what is the total amount of money the customers had before ordering?"*. To process this query, we need to join the two tables *CUSTOMER* and *ORDERS* in the TPC-H benchmark, and then sum up the two fields *o_totlaprice* and *c_acctbal*. Since SnappyData does not support sampling over the join operation, in this experiment it first executes the join operation between the two tables *CUSTOMER* and *ORDERS*, then performs the sampling over the join output, and finally calculates the sum of the two fields *o_totalprice* and *c_acctbal*.

Figure 11(b) presents the latencies of ApproxJoin and SnappyData in processing the aforementioned query with varying sampling fractions. SnappyData has a significantly higher latency than ApproxJoin, simply because it performs sampling only after the join operation finishes. For example, with a sampling fraction of

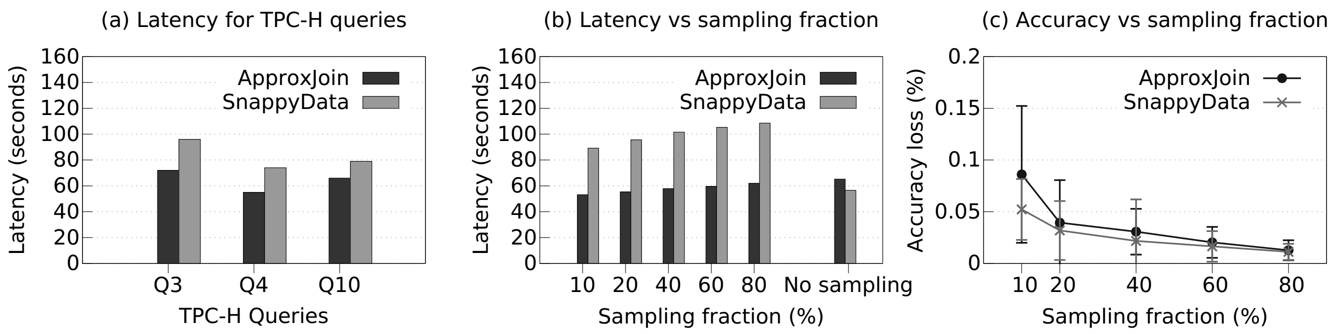

Figure 11: Comparison between ApproxJoin and the state-of-the-art SnappyData system in terms of (a) latency with different TPC-H queries, (b) latency, and (c) accuracy with different sampling fractions.

60%, SnappyData achieves a 1.77× higher latency than ApproxJoin, even though it is faster when both systems do not perform sampling (i.e., sampling fraction is 100%). Note however that, sampling is inherently needed when one handles joins with large-scale inputs that require a significant number of cross-product operations. Figure 11(c) shows the accuracy losses of ApproxJoin and SnappyData. ApproxJoin achieves an accuracy level similar to SnappyData. With a sampling fraction of 60%, ApproxJoin achieves an accuracy loss of 0.021%, while SnappyData achieves an accuracy loss of 0.016%.

6 EVALUATION: REAL-WORLD DATASETS

We evaluate ApproxJoin using two real-world datasets: (a) network traffic monitoring, and (b) Netflix Prize.

6.1 Network Traffic Monitoring Dataset

Dataset. We use the CAIDA network traces [20] which were collected on the Internet backbone links in Chicago in 2015. In total, this dataset contains 115, 472, 322 TCP flows, 67, 098, 852 UDP flows, and 2, 801, 002 ICMP flows. Here, a flow denotes a two-tuple network flow that has the same source and destination IP addresses.

Query. We use ApproxJoin to process the query: *What is the total size of the flows that appeared in all TCP, UDP and ICMP traffic?* To answer this query, we need to perform a join operation across TCP, UDP and ICMP flows.

Results. Figure 12(a) first shows the latency comparison between ApproxJoin (with filtering but without sampling), Spark repartition join, and native Spark join. ApproxJoin achieves a latency 1.72× and 1.57× lower than Spark repartition join and native Spark join, respectively. Interestingly, native Spark join achieves a lower latency than Spark repartition join. This is because the dataset is distributed quite uniformly across worker nodes in terms of the join-participating flow items, i.e., there is little data skew. Figure 12(a) also shows that ApproxJoin significantly reduces the shuffled data size by a factor of 300× compared with the two Spark join systems.

Next, different from the experiments in §5, we extend Spark repartition join by enabling it to sample the dataset before the actual join operation. This leads to the lowest latency it could achieve. Figure 12(b) shows that ApproxJoin achieves a similar latency even to this extended Spark repartition join. In addition, Figure 12(c) shows the accuracy loss comparison between ApproxJoin and Spark repartition join with different sampling fractions. As the sampling fraction increases, the accuracy losses of ApproxJoin and Spark repartition join decrease, but not linearly. ApproxJoin produces around 42× more accurate query results than the Spark repartition join with the same sampling fraction.

6.2 Netflix Prize Dataset

Dataset. We also evaluate ApproxJoin based on the Netflix Prize dataset which includes around $100M$ ratings of 17, 770 movies by 480, 189 users. Specifically, this dataset contains 17, 770 files, one per movie, in the *training_set* folder. The first line of each such file contains *MovieID*, and each subsequent line in the file corresponds to a rating from a user and the date, in the form of $\langle UserID, Rating, Date \rangle$. There is another file $qualifying.txt$ which contains lines indicating $MovieID$, $UserIDs$, and the rating $Dates$.

Query. We perform the join operation between the dataset in *training_set* and the dataset in $qualifying.txt$ to evaluate ApproxJoin in terms of latency. Note that, we cannot find a meaningful aggregation query for this dataset; therefore, we focus on only the latency but not the accuracy of the join operation.

Results. Figure 12(a) shows the latency and the shuffled data size of ApproxJoin (with filtering but without sampling), Spark repartition join, and native Spark join. ApproxJoin is 1.27× and 2× faster than Spark repartition join and native Spark join, respectively. The result in Figure 12(a) also shows that ApproxJoin reduces the shuffled data size by 3× and 1.7× compared with Spark repartition join and native Spark join, respectively. In addition, Figure 12(b) presents the latency comparison between these systems with different sampling fractions. For example, with the sampling fraction of 10%, ApproxJoin is 6× and 9× faster than Spark repartition join and native Spark join, respectively. Even without sampling (i.e., sampling fraction is 100%), ApproxJoin is still 1.3× and 2× faster than Spark repartition join and native Spark join, respectively.

7 RELATED WORK

Over the last decade, approximate computing has been widely applied in data analytics systems [5, 6, 8, 29, 30, 32, 40, 41, 43–46, 52, 53]. Various approximation techniques have been proposed to make trade-offs between required resources and output quality, including sampling [9, 25], sketches [23], and online aggregation [27, 38].

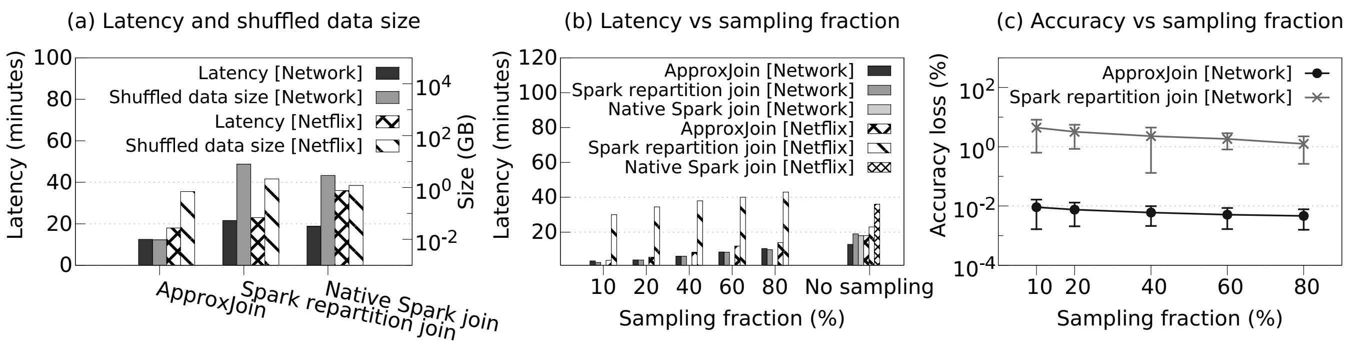

Figure 12: Comparison between ApproxJoin, Spark repartition join, and native Spark join based on two real-world datasets: (1) Network traffic monitoring dataset (denoted as [Network]), and (2) Netflix Prize dataset (denoted as [Netflix]).

Chaudhari et al. provide a sampling over join mechanism by taking a sample of an input and considering all statistical characteristics and indices of other inputs [22]. AQUA [6] system makes use of simple random sampling to take a sample of joins of inputs that have primary key-foreign key relations. BlinkDB [8] proposes an approximate distributed query processing engine that uses stratified sampling [9] to support ad-hoc queries with error and response time constraints. VerdictDB [39], SnappyData [47], and SparkSQL [11] adopt the approximation techniques from BlinkDB to support approximate queries. Quickr [5] deploys distributed sampling operators to reduce execution costs of parallel, ad-hoc queries that may contain multiple join operations. Quickr first injects sampling operators into the query plan and searches for an optimal query plan among sampled query plans to execute input queries. Unfortunately, all of these systems require a priori knowledge of the inputs. For example, AQUA [6] requires join inputs to have primary key-foreign key relations. For another example, the sampling over join mechanism in [22] needs the statistical characteristics and indices of inputs. Finally, BlinkDB [8] utilizes the most frequently used column sets to perform offline stratified sampling over them. Afterwards, the samples are cached, such that queries can be served by selecting the relevant samples and executing the queries over them. While useful in many applications, BlinkDB and these other systems cannot process queries over new inputs, where queries or inputs are typically not known in advance.

Ripple Join [26] implements online aggregation for joins. Ripple Join repeatedly takes a sample from each input. For every item selected, it is joined with all items selected in other inputs so far. Recently, Wander Join [34] improves over Ripple Join by performing random walks over the join data graph of a multi-way join. However, their approach crucially depends on the availability of indices, which are not readily available in "big data" systems like Apache Spark. In addition, the current Wander Join implementation is single-threaded, and parallelizing the walk plan optimization procedure is non-trivial. In this work, we proposed a simple but efficient sampling mechanism over joins which works not only on a single node but also in a distributed setting.

Our work builds on recent advancements in approximate computing for stream analytics [32, 41, 44, 46, 52]. More specifically, IncApprox [32] is a stream analytics system that combines approximate computing and incremental computing [13, 15, 16]. StreamApprox [45, 46] designs a distributed sampling algorithm to take "on-the-fly" samples of the input data stream. ApproxIoT [52] extends StreamApprox to supports approximate stream data analytics in the IoT infrastructure. Finally, PrivApprox [43, 44] makes use of a combination of randomized response and approximate computing to support privacy-preserving stream analytics. However, all of these systems currently do not support approximate joins.

8 CONCLUSION

In spite of decades of research interest in approximate query processing, the problem of approximating statistical properties of the join output remains challenging [4, 21, 31, 36]. In this work, we address some of the challenges associated with performing approximate joins for distributed data analytics systems. By performing sampling during the join operation, we achieve low latency as well as high accuracy. In particular, we employ a sketching technique (i.e., Bloom filters) to reduce the size of the shuffled data during a join and we construct a stratified sample during the join in a distributed setting. We implemented our techniques in a system called ApproxJoin using Apache Spark and evaluated its effectiveness using a series of microbenchmarks and real-world workloads. Our evaluation shows that ApproxJoin significantly reduces query response time as well as the data shuffled through the network, without losing the accuracy of the query results compared with the state-of-the-art systems.

Supplementary material. We provide the analysis of ApproxJoin covering both communication and computation complexities; and also discuss three alternative design choices for Bloom filters in the technical report [42].

Software availability. The source code of ApproxJoin is publicly available: https://ApproxJoin.github.io/

Acknowledgments. This work is supported by the European Unions Horizon 2020 research and innovation programme under grant agreements No. 777154 (ATOMSPHERE) and No. 780681 (LEGaTO), the Alan Turing Institute, and an Amazon AWS Research grant. This material is based upon work supported by the National Science Foundation under Grants No. CCF-1816577 and CCF-1747447.

REFERENCES

[1] Apache Flink. https://flink.apache.org/. Accessed: August, 2018.
[2] Apache Hadoop. http://hadoop.apache.org/. Accessed: August, 2018.
[3] Apache Spark. https://spark.apache.org. Accessed: August, 2018.
[4] Approximate query processing where do we go from here? http://wp.sigmod.org/?p=2183. Accessed: August, 2018.
[5] Quickr: Lazily Approximating Complex Ad-Hoc Queries in Big Data Clusters. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2016.
[6] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1999.
[7] 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.
[8] 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.
[9] 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.
[10] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.
[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2015.
[12] R. Barber, G. Lohman, I. Pandis, V. Raman, R. Sidle, G. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2014.
[13] P. Bhatotia. *Incremental Parallel and Distributed Systems*. PhD thesis, Max Planck Institute for Software Systems (MPI-SWS), 2015.
[14] 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.
[15] 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.
[16] 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.
[17] 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.
[18] 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.
[19] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 1970.
[20] CAIDA. The CAIDA UCSD Anonymized Internet Traces 2015 (equinix-chicago-dirA). http://www.caida.org/data/passive/passive_2015_dataset.xml.
[21] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2017.
[22] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1999.
[23] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 2012.
[24] A. Doucet, S. Godsill, and C. Andrieu. On sequential monte carlo sampling methods for bayesian filtering. *Statistics and Computing*, 2000.
[25] 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.
[26] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 1999.
[27] 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.
[28] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American statistical Association*, 1952.
[29] N. Kamat and A. Nandi. Perfect and maximum randomness in stratified sampling over joins. *CoRR*, 2016.
[30] N. Kamat and A. Nandi. A unified correlation-based approach to sampling over joins. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2017.
[31] T. Kraska. Approximate query processing for interactive data science. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.
[32] D. R. Krishnan, D. L. Quoc, P. Bhatotia, C. Fetzer, and R. Rodrigues. IncApprox: A Data Analytics System for Incremental Approximate Computing. In *proceedings of International Conference on World Wide Web (WWW)*, 2016.
[33] T. Lee, K. Kim, and H.-J. Kim. Join processing using bloom filter in mapreduce. In *Proceedings of the 2012 ACM Research in Applied Computation Symposium (RACS)*, 2012.
[34] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*, 2016.
[35] S. Lohr. *Sampling: design and analysis*. Cengage Learning, 2009.
[36] B. Mozafari. Approximate query engines: Commercial challenges and research opportunities. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*, 2017.
[37] S. Natarajan. *Imprecise and Approximate Computation*. Kluwer Academic Publishers, 1995.
[38] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2011.
[39] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2018.
[40] N. Potti and J. M. Patel. DAQ: A New Paradigm for Approximate Query Processing. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2015.
[41] D. L. Quoc. *Approximate Data Analytics Systems*. PhD thesis, Technische Universität Dresden (TU Dresden), 2017.
[42] D. L. Quoc, I. E. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe. Approximate distributed joins in apache spark. *CoRR*, abs/1805.05874, 2018.
[43] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. Privacy preserving stream analytics: The marriage of randomized response and approximate computing. *CoRR*, abs/1701.05403, 2017.
[44] D. L. Quoc, M. Beck, P. Bhatotia, R. Chen, C. Fetzer, and T. Strufe. PrivApprox: Privacy-Preserving Stream Analytics. In *Proceedings of the 2017 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC)*, 2017.
[45] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. Approximate Stream Analytics in Apache Flink and Apache Spark Streaming. *CoRR*, abs/1709.02946, 2017.
[46] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe. StreamApprox: Approximate Computing for Stream Analytics. In *Proceedings of the International Middleware Conference (Middleware)*, 2017.
[47] J. Ramnarayan, B. Mozafari, S. Wale, S. Menon, N. Kumar, H. Bhanawat, S. Chakraborty, Y. Mahajan, R. Mishra, and K. Bachhav. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, 2016.
[48] S. K. Thompson. *Sampling*. Wiley Series in Probability and Statistics, 2012.
[49] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - A petabyte scale data warehouse using Hadoop. In *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*, pages 996–1005, 2010.
[50] Y. Tian, F. Özcan, T. Zou, R. Goncalves, and H. Pirahesh. Building a hybrid warehouse: Efficient joins between data stored in hdfs and enterprise warehouse. *ACM Trans. Database Syst.*, 2016.
[51] Y. Tian, T. Zou, F. Ozcan, R. Goncalves, and H. Pirahesh. Joins for hybrid warehouses: Exploiting massive parallelism in hadoop and enterprise data warehouses. In *In Proceedings of the 2015 International Conference on Extending Database Technology (EDBT)*, pages 373–384, 2015.
[52] Z. Wen, D. L. Quoc, P. Bhatotia, R. Chen, and M. Lee. ApproxIoT: Approximate Analytics for Edge Computing. In *Proceedings of the 38th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2018.
[53] F. Yu, W.-C. Hou, C. Luo, D. Che, and M. Zhu. CS2: A New Database Synopsis for Query Estimation. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD)*, 2013.
[54] 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.
[55] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2008.