Plexus: Optimizing Join Approximation for Geo-Distributed Data Analytics

Joel Wolfrath
University of Minnesota
Minneapolis, MN, USA
wolfr046@umn.edu

Abhishek Chandra
University of Minnesota
Minneapolis, MN, USA
chandra@umn.edu

ABSTRACT

Modern applications are increasingly generating and persisting data across geo-distributed data centers or edge clusters rather than a single cloud. This paradigm introduces challenges for traditional query execution due to increased latency when transferring data over wide-area network links. Join queries in particular are heavily affected, due to their large output size and amount of data that must be shuffled over the network. Join sampling—computing a uniform sample from the join results—is a useful technique for reducing resource requirements. However, applying it to a geo-distributed setting is challenging, since acquiring independent samples from each location and joining on the samples does not produce uniform and independent tuples from the join result. To address these challenges, we first generalize an existing join sampling algorithm to the geo-distributed setting. We then present our system, Plexus, which introduces three additional optimizations to further reduce the network overhead and handle network and data heterogeneity: (i) weight approximation, (ii) heterogeneity awareness and (iii) sample prefetching. We evaluate Plexus on a geo-distributed system deployed across multiple AWS regions, with an implementation based on Apache Spark. Using three real-world datasets, we show that Plexus can reduce query latency by up to 80% over the default Spark join implementation on a wide class of join queries without substantially impacting sample uniformity.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11…$15.00
https://doi.org/10.1145/3620678.3624643

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Information systems** → **Join algorithms**; **Query optimization**.

KEYWORDS

Join Algorithms, Distributed Systems, Query Optimization, Wide Area Network

ACM Reference Format:
Joel Wolfrath and Abhishek Chandra. 2023. Plexus: Optimizing Join Approximation for Geo-Distributed Data Analytics. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3620678.3624643

1 INTRODUCTION

Large organizations collect and persist vast amounts of data in a *geo-distributed* manner [4, 11]. For example, Alibaba manages tens of geo-distributed data centers (DCs), each containing millions of tables, which are used to process millions of cross-DC analytical jobs every day [11]. Persisting data in edge clusters or data centers close to the end users provides several benefits, including low latency response times for users and compliance with data regulation laws. However, this trend presents challenges for efficient data analytics, especially when join queries are involved as they shuffle sizable amounts of data over the wide area network and generate very large outputs [23]. Generating a random sample from a join and performing analytics against the sample rather than the full join result is a cost-effective way to address this limitation [10]. If the tuples in the join result are sampled *uniformly and independently*, they can be used for a wide variety of analytical tasks, including estimating aggregate functions and building machine learning models for regression, classification, etc. Sampling can also handle arbitrary selection predicates, which makes it a desirable approach for approximating query results. However, when data is geo-distributed in nature, existing join approximation algorithms become increasingly I/O bound. In an initial experiment, we observed 3-4x performance penalty when computing approximate joins with geo-distributed tables rather than tables located in the same regional data center.

Shuffling large tables over the wide-area network (WAN) causes the geo-distributed implementation to scale poorly as the data size increases. This illustrates the need for more efficient join sampling algorithms in the wide-area.

A substantial amount of research has been conducted to investigate join sampling algorithms, but these algorithms are designed to run in a centralized fashion [5, 7, 17, 22, 29, 38]. These techniques are useful starting points for exploring the geo-distributed setting, but are not scalable *per se* since the designs emphasize CPU efficiency rather than I/O efficiency. Systems designed for distributed join computation are also prevalent, but the distributed aspects are confined to a single location [3, 21, 26]. The task is to optimize join computation or sampling *on a cluster operating within a single data center*. These approaches are generally still required to shuffle all tuples participating in the join over the network, which introduces a large penalty when high-speed local networks are swapped for the slower, more heterogeneous WAN.

Geo-distributed analytics systems have introduced optimizations for addressing constraints imposed by the WAN, including the optimal placement of tasks [24] and scheduling techniques to reduce query latency or WAN traffic [25, 32, 33]. However, these frameworks are agnostic to the operation and data characteristics, which are important to exploit. Furthermore, in the case of join computation, these techniques often generate wide-area traffic linear in the number of tuples being joined, which may be extremely large [24]. Join query optimization has been proposed in a distributed setting, but it is limited to joins that evaluate an aggregate function [14]. We argue that approximate joins are a better fit for the wide-area since the amount of shuffled data has the potential to be much smaller. While it is trivial to produce a sample from a join while generating less WAN traffic, it is challenging to design an algorithm that can reduce traffic *and generate a uniform sample*, i.e. ensure that each tuple in the full join result is sampled with equal probability. To address these challenges, we generalize a centralized join sampling algorithm to the geo-distributed setting while maintaining sample uniformity. We then propose additional optimizations to further reduce the network traffic and address the heterogeneity present in the geo-distributed deployment.

Contributions. We make the following contributions:

- We generalize an existing join sampling algorithm in non-trivial ways to adapt to the geo-distributed setting.
- We present Plexus, a system that introduces three optimizations to the geo-distributed join sampling algorithm to further reduce the network overhead and handle network and data heterogeneity: (i) weight approximation, (ii) heterogeneity awareness and (iii) sample prefetching.
- We explore a key trade-off in join sampling between sample or data uniformity and query latency.
- We implement Plexus on a Spark-based analytics platform, and evaluate it across a set of geo-distributed sites.
- Using three real-world datasets, we show that Plexus can reduce query latency by up to 80% over the default Spark join implementation on a wide class of join queries with negligible impact to sample uniformity.

We show that Plexus is a more scalable alternative for computing joins in the wide-area and can substantially reduce query latency using the above approaches.

2 PRELIMINARIES

2.1 Join Sampling

Join operations are a fundamental building block for analytics over relational data. However, computing a join can be expensive; in the worst case, computing a join over a set of tables $T_1, T_2, \ldots, T_n$ involves producing the Cartesian product of the selected tables, which generates $\prod_i \mid T_i \mid$ tuples in the result, where $\mid T_i \mid$ is the cardinality of T_i. Approximating the join is an appealing alternative, since it avoids much of the computation and shuffling required to produce the full result. While some approximation techniques focus on estimating specific functions of the data, such as aggregates [7, 17, 26], we focus on sampling from joins, a general approximation technique that can be used for a wide variety of analytical tasks. Generating uniform samples from a join is a challenging problem, since it is insufficient to simply collect samples from each table and join on the samples [10]. For example, consider a join of two tables, $T_1 \bowtie T_2$, with keys $\{k_0, k_1\}$ in T_1 and keys $\{k_0, k_2\}$ in T_2. If we sample individual tables and only obtain k_1 keys from T_1 and k_2 keys from T_2, then the resulting join is the empty set. For sampling to work, *tuples must be selected with a probability proportional to their frequency in the join result.* Sampling tables directly produces independent *but not uniform* samples from the join result and requires large sample sizes [16].

The task is further complicated by considering properties of geo-distributed data sources. We assume that tables participating in the join are distributed geographically across multiple sites [4, 11]. The main challenge is the difference in performance between the WAN and a data center network. The WAN is substantially slower and more heterogeneous, which creates a bottleneck for existing sampling algorithms. Reducing query latency in this setting largely depends on making efficient use of the network.

2.2 Problem Statement

We consider a geo-distributed system composed of a set of tables $\{T_i \mid i \in 1..n\}$ partitioned arbitrarily across a set of sites $\{s_j \mid j \in 1..m\}$. For a given table T_i, we use the notation $T_{i,j}$ to denote the subset of the table T_i that exists at site j. Figure 1 shows our system model, with $n = 3$ tables

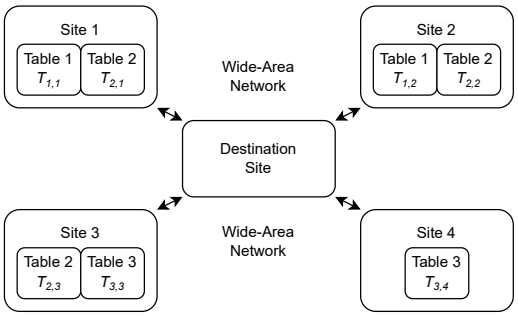


Figure 1: System model.

partitioned across $m = 4$ sites. We assume that the tables consist of at least one *key* column, which will be used to join tables, along with one or more value columns. The task is to produce *uniform and independent* samples from the multi-way (equi-)join $T_1 \bowtie \ldots \bowtie T_n$. Furthermore, we seek to minimize the query latency associated with computing these samples. While our objective is to obtain *uniform and independent* samples from the join result, we also explore options for relaxing the uniformity requirement in an effort to reduce query latency.

2.3 Centralized Sampling with Exact Weights

The recent join sampling work by Zhao *et al.* [38] proposes an algorithm called "Exact Weights" which computes uniform and independent samples from join results. We briefly discuss their centralized join sampling algorithm, before generalizing it to the geo-distributed setting.

To realize uniform and independent samples from the join, we need to associate a *weight* with each tuple and perform *weighted random sampling* from the original tables using probabilities proportional to the weights [38]. Two sequential passes over the participating tables are required to compute this sample. The first pass performs weight generation and aggregates weights for the keys in each table, which are needed to maintain sample uniformity. The second pass performs sample collection from each table using these weights. We begin by defining weights in this context.

Definition 2.1. The weight $w(t)$ associated with tuple t in T_i, is the frequency of that tuple in the (partial) join result $T_i \bowtie T_{i+1} \bowtie \ldots \bowtie T_n$. We also define $w(t) = 1$ for all $t \in T_n$.

Weight Generation. The objective of weight generation is to compute $w(t)$ for each tuple. Note that our definition implies that the weights for T_1 represent the frequency of each tuple *in the full join result*, since they are computed for $T_1 \bowtie \ldots \bowtie T_n$. Weight generation makes a backward pass over the tables, beginning at T_n (where $w(t) = 1$ for all t) and terminating at T_1.

Definition 2.2. For a given table T_i and join key k, define $w_k = \sum w(t)$ for all tuples $t \in T_i$ with join key k.

Using this definition, we observe that the weight generated for key k at a given table T_i is equal to the value of w_k in the previous table (which is T_{i+1} since we are making a backward pass). To illustrate this process, we consider an example task of generating weights over three tables, as shown in figure 2a. To build up the weights, the algorithm makes a backward pass over the tables (i.e. starting with T_3 and terminating at T_1). The weights stored in each table only represent the frequency of that tuple *in a join with the tables already visited.* So the weights in T_2 represent the frequency of that tuple in a join with T_3 and the weights in T_1 will represent the frequency in the full join $T_1 \bowtie T_2 \bowtie T_3$. Tuples in T_3 are defined to have a weight of 1, *which allows weights in subsequent tables to simply be w_k, the sum of weights with matching keys in the previous table.* These weights can be computed iteratively or with dynamic programming.

Sample Collection. Once we have weights, we can make a forward pass over the tables and perform weighted random sampling with replacement to produce a uniform sample from the join. For example, assume we wish to collect two samples from the join using the weights computed in figure 2a. Beginning with T_1, we convert the weights to probabilities, which yields 0.25 for each A key tuple and 0.5 for the B key, as illustrated in figure 2b. If our random sampling yields one A key and one B key (indicated by the gray color in figure 2b), we will now move to table T_2 and randomly select a single sample from each key obtained from T_1. The weights are converted to sampling probabilities within each key group, i.e. by computing $w(t)/w_k$. Since there is only one A key, we select that sample from T_2. For the B key, we have two tuples each with weight 1, so we assign probability 1/2 to both tuples and draw one. The final table only has one tuple for each key, so no random sampling is required. Concatenating these sampled tuples from each table produces a uniform sample from the join. Note that key C does not appear in the final join result, since its weight will never propagate to T_2 or T_1 during weight generation.

3 GEO-DISTRIBUTED JOIN SAMPLING

Having described the centralized Exact Weights sampling algorithm, we now generalize each of the phases to the geo-distributed setting. Algorithm 1 shows the pseudocode for our geo-distributed exact weights algorithm.

(a) Weight generation phase. **(b) Sample collection phase.**

Figure 2: Centralized Sampling with Exact Weights over three tables.

Weight Generation. In the geo-distributed setting, tables can be partitioned across multiple sites. We can utilize the dynamic programming approach for the geo-distributed case, but we require additional steps to aggregate the weights from each site prior to computing weights for the next table.

Definition 3.1. For a given table and join key k, we define $w_{k,i}$ to be the sum of weights for all tuples at site i that have join key k.

It follows immediately from this definition that we can compute $w_k = \sum_i w_{k,i}$, which computes the total weight for a given key across sites (consistent with definition 2.2). Following the tables in Figure 2, the geo-distributed weight generation process begins with the last table in the join, T_3, assigning a value of 1 to each tuple. We then query (in parallel) the local weights for all edges that contain $T_{3,i}$, which gives us $w_{k,i}$ for each site. This is followed by summing the weights for any keys that exist at multiple sites (which produces w_k) and sending the resulting aggregated weights (w_k for all k) to each site that contains $T_{2,i}$. This step is repeated to generate weights for T_1, after which all tuples have the correct sampling weights needed to generate uniform samples from the join result. The first part of Algorithm 1 outlines this flow in the geo-distributed setting.

Sample Collection. Once exact weights have been computed for each partitioned table, we can proceed with the sample collection phase. The second part of Algorithm 1 outlines the sample collection phase at a high level. This phase begins by first partitioning the sample size according to the sum of the tuple weights at each site, which is required to maintain sample uniformity. Once we have a sample size for each site, we query (in parallel) all sites containing $T_{1,i}$ (the first table in the join), where tuples are drawn by weighted random sampling with replacement. Once the first table has been sampled, the resulting keys from each site are counted and form the sample sizes for keys in the next table. When the next table is sampled, weighted random sampling with replacement is performed at the key level. As an example, assume two sites s_1 and s_2 have tuples with a given key k and the sum of weights for those tuples is $w_{k,1} = 5$ and $w_{k,2} = 15$ at each site respectively, which yields $w_k = 20$. Then, for each sample, we first select a site at random, with s_1 having probability 0.25 and s_2 having probability 0.75, then a tuple can be sampled from the selected site. In practice, we can use a multinomial distribution to randomly generate samples sizes for each site in one operation, then samples can be collected from each site in parallel.

Zhao *et al.* [38] showed that using the Exact Weights algorithm to perform weighted random sampling with replacement will result in a uniform sample from the join result; what remains to be shown is that collecting samples from sites proportional to their key weight maintains uniformity.

Theorem. *The geo-distributed exact weights algorithm generates a uniform sample from the full join result.*

Proof. To maintain uniformity, an arbitrary tuple t with join key k must be selected with probability proportional to its generated weight[38], i.e. with probability $w(t)/w_k$. In the geo-distributed case, each tuple exists at a given site, so to be selected, the site must be independently sampled first, then the tuple sampled using the local weights at the site. A tuple t with key k at site i is sampled with probability $(w_{k,i}/w_k)$ x $(w(t)/w_{k,i})$ which reduces to $w(t)/w_k$. Therefore, the resulting sample is uniform.

4 PLEXUS: OPTIMIZING GEO-DISTRIBUTED JOIN SAMPLING

The geo-distributed exact weights algorithm (Algorithm 1) has some inherent scaling limitations. First, aggregating weights for each tuple/key and collecting samples can be network intensive. The algorithm is also sequential as it systematically generates weights and collects samples. Furthermore, the network can be heterogeneous in terms of the link bandwidths of the sites and the data distribution across the sites can also be non-uniform. We now present our geo-distributed join sampling framework called *Plexus* that is designed to overcome these limitations by introducing various optimizations to the exact weights algorithm.

Algorithm 1: Geo-Distributed Exact Weights

```
Input: List of sites and tables, Sample Size, Target Query
Result: Uniform and Independent Join Samples

// Weight Generation
previousWeights ← []
for table in reverse(tables) do
    for site in sites do
        if table exists at site then
            generateWeightsAsync(site, table, previousWeights)
        end
    end
    blockWaitingForWeights()
    previousWeights ← mergeWeightsForTable(table)
end

// Sample Collection
joinSample ← []
keys ← []
for table in tables do
    for site in sites do
        if table exists at site then
            sizes ← getSampleSizes(site, table, keys)
            collectSamplesAsync(site, table, sizes)
        end
    end
    blockWaitingForSamples()
    samples ← aggregateSamplesForTable(table)
    keys ← getKeysForNextTable(samples)
    joinSample ← concatenate(joinSample, samples)
end

return joinSample
```

4.1 Weight Approximation

The weights generated for every table at each site are a major source of network traffic. The exact weights algorithm generates traffic that is linear in the number of keys, which could equal the number of tuples in the entire table in the worst case. Generating this amount of traffic could substantially increase the overall query execution time.

One option for reducing traffic would be to transfer a sketch of the weights, rather than the actual weights. A Count-Min sketch could provide approximate weights and uses space that is sublinear in the number of keys [6]. However, using this kind of sketch presents a few problems for weight approximation: (1) false positives could cause us to sample keys not actually present in the join, (2) the upper bound on the error for each weight is linear in the number of tuples, and (3) the sketch must be the same size for each site to be merged, regardless of the number of weights at each site. For these reasons, we consider an alternative approach for weight approximation.

To address all three shortcomings of weight sketching, we propose sending only a random sample of the weights over the wide-area network. For a given table and join key k, let $w_{k,i}$ denote the true value of k's weight at site i. We assign a probability $p_{k,i}$ to each weight and perform Poisson sampling to obtain a subset of the weights to send over the WAN. The sampling probabilities are generated using an existing algorithm [37] which computes $p_{k,i} = w_{k,i}^2/(w_{k,i}^2 + C_i)$, for a constant C_i that depends on the sample size for site i. Weights that are not sampled are assumed to be zero. If a weight is sampled, we send the estimate $\hat{w}_{k,i} = w_{k,i}/p_{k,i}$ over the network, rather than the true value of the weight. Sampling with these probabilities and using $\hat{w}_{k,i}$ as an estimator for the true weight is known to be an unbiased sampling scheme [37]. Furthermore, it follows immediately that $\hat{w}_k = \sum_i \hat{w}_{k,i}$ is an unbiased estimator for w_k, due to the linearity of expectation. This is an important insight, since unbiasedness guarantees the weights produced at each site are correct in expectation. The join samples generated by our system will be consistent (in expectation) with the true distribution of keys in the full join result and the weights in the final table can also be used for unbiased join size estimation.

Weight Sample Size Estimation. To use sampling as our weight approximation technique, we require a mechanism for determining how many weights $w_{k,i}$ to transfer over the WAN from site i. We define a weight error bound ϵ which is used to bound the average standard error for the local weights. For all keys k at site i, we require a sample size large enough, such that:

$$\frac{1}{|\,k\,|}\sum_{k}\sqrt{\mathrm{Var}[\hat{w}_{k,i}]} \;\leq\; \epsilon \tag{1}$$

where $|\,k\,|$ is the number of unique keys at site i. This design allows for more samples to be assigned to sites with more weights or larger weights, which was a limitation associated with sketching. However, there is no way to allow the sample size to scale with the network or compute capacity at each site. This would require *a-priori* knowledge of which keys in the join result exist at each site, which we assume is unknown until the weight generation phase is completed. We explore alternative ways of leveraging heterogeneity in the system in section 4.2.

4.2 Heterogeneity Awareness

There are two main sources of heterogeneity in our setting: network heterogeneity and data heterogeneity. We explore methods to improve performance by exploiting heterogeneity while controlling for the effect this may have on query accuracy. This analysis assumes an awareness of the network and data heterogeneity present in the system; we discuss

mechanisms for estimating these quantities in section 4.3.

Network Heterogeneity. An important property of geo-distributed systems is the network heterogeneity, both in terms of average throughput and variance. While sampling reduces the overhead of a full join computation, sending samples over the network still generates a non-trivial amount of network traffic, which impacts query latency in the wide-area. While we cannot avoid sending this data, we can attempt to reduce the query latency by exploiting the network heterogeneity.

During sample collection, we can reduce the overall query latency by requesting more data from the lower-latency sites, assuming the same keys are present at multiple sites. If we simply seek to minimize latency, we can solve the following optimization problem to determine how many samples to collect from each site:

$$\begin{aligned} \min_{n} \quad & \max(\ell(n_1), \ell(n_2), \dots, \ell(n_m)) \\ \text{s.t.} \quad & \sum_{i=1}^{m} n_i = S \end{aligned} \tag{2}$$

where S is the total number of samples requested across all sites and $\ell(n_i)$ is the estimated latency required to transfer n_i samples from site i. We further assume that the latency has the form $\ell(n_i) = n_i \ / \ b_i$, where b_i represents the bandwidth for the connection to site i. Then the maximum latency (makespan) is minimized by setting the sample sizes proportional to the expected throughput for each site, i.e. by setting $n_i = S\ (b_i / \sum_j b_j)$.

Data Heterogeneity. Deriving sample sizes proportional to each site's latency will minimize overall query latency and preserve the distribution of join keys, but it has the potential to bias queries that depend on other columns in the tables. If we make a strong assumption that the tuples are *independent and identically distributed* (IID) across all sites, then skewing sample sizes in favor of performance has no effect. However, this assumption is unrealistic in practice. Since sites may exist in different time zones and collect data from different populations, it is reasonable to assume that some heterogeneity is present across sites and needs to be accounted for.

We assume each site has data following a unique data distribution. Therefore, a given column C in the join result can be modeled as a random variable with a mixture distribution, i.e. $C \sim \sum_i \theta_i F_i(x)$ where each F_i is the cumulative distribution function[1] (CDF) and $\theta_i = n_i \ / \ \sum_j n_j$ is the fraction of samples obtained from site i. We can then add the following constraint to the optimization problem in equation 2:

$$\max \mid \sum_i \theta_i F_i(x) - F(x) \mid \ \leq \ \delta \tag{3}$$

which requires that the maximum absolute distance between the heterogeneity aware CDF and the true mixture CDF, $F(x)$, for the column C in the join result is at most δ. This bound on the performance bias is similar to how the KS-test works for measuring distribution similarity, and δ can be selected in a way that corresponds to the KS-test at a specified significance level. Requiring the user to specify δ rather than simply running a KS-test forces a selection of an *effect size*, and avoids the issues where the test detects differences too small to matter [18]. Figure 3 illustrates how this optimization works in practice. The two dotted CDFs represent the data distribution at two separate sites, one designated a fast site and the other slow. Data is drawn from a normal distribution, with $\mu = 10$ for the fast site and $\mu = 20$ for the slow site (σ is fixed at 5 for both sites). To minimize latency, we have a preference for collecting more data from the fast site (and skewing the CDF in that direction). The solid line represents the true mixture distribution when data is selected uniformly from each site (i.e. $\theta_{fast} = \theta_{slow} = 0.5$). If we set $\delta = 0.1$, the dashed blue line represents the degree we are allowed to bias the performance toward the faster site. The closer we get to the faster site CDF, the smaller the resulting query latency.

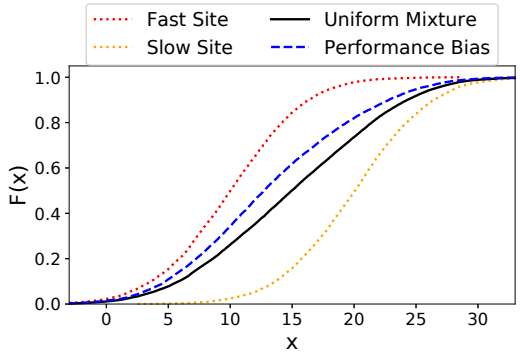


Figure 3: A mixture distribution can be skewed toward the fast site with a bound δ.

Adding this constraint to the optimization greatly complicates the problem and makes it more expensive to solve. This is caused by the fact that we must use empirical distribution functions in practice, not theoretical ones, so we need to iterate over our estimates to evaluate them. We use a hill climbing approach in Plexus for solving the problem (algorithm 2).

[1]This formulation focuses on continuous variables, but alternatives are available for the discrete case, without loss of generality.

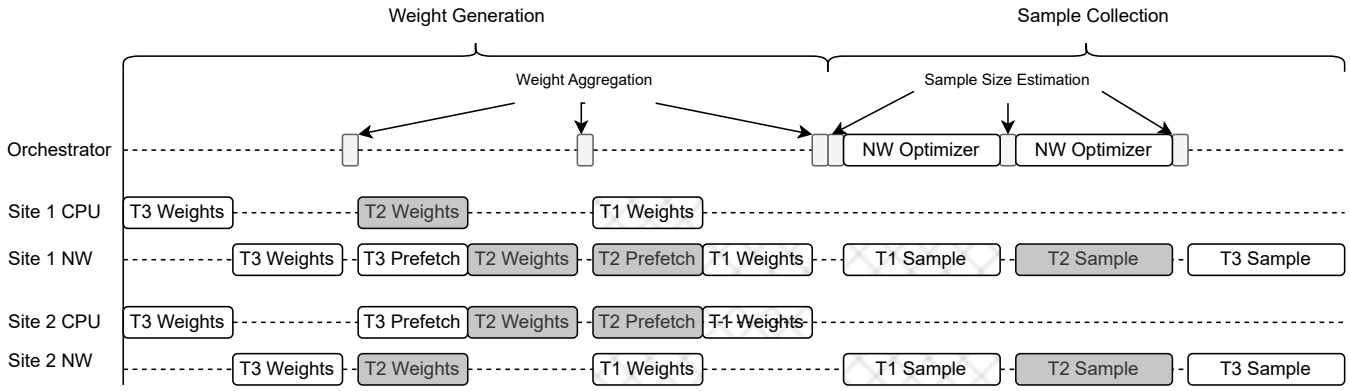


Figure 4: The Plexus pipeline. Wherever possible, network intensive operations are executed in parallel with compute intensive operations to avoid resource contention.

The algorithm begins by computing the latency-optimal allocation to determine if that satisfies the CDF constraint. This requires solving the unconstrained problem (eq. 2) which is efficient and only requires computation linear in the number of sites. If the latency-optimal allocation is not a feasible solution, we begin the hill-climbing approach, initialized with the point in the domain that represents the true mixture distribution. At each optimization step, we select two sites at random and determine which site has a lower sampling latency. We then attempt to move a proportion of the requested sample size (α) from the slower site (s_{slow}) to the faster site (s_{fast}). More specifically, we need to select the maximum value $\alpha \in [0, \theta_{slow}]$ such that for all points in the CDF domain, α satisfies:

$$\mid (\theta_{fast} + \alpha)F_{fast} + (\theta_{slow} - \alpha)F_{slow} + \sum_{\substack{i \neq fast \\ i \neq slow}} \theta_i F_i - F \mid \leq \delta$$

which allows α to represent the proportion of the sample allocation we can move from the high latency site to the low latency site. Computing α can be done analytically by solving the quadratic equation:

$$\left(\alpha_x[F_{fast} - F_{slow}] + \sum_i \theta_i F_i - F\right)^2 - \delta^2 = 0 \qquad (4)$$

for each point in the CDF domain and setting $\alpha = \min\{\alpha_x\}$, which requires time linear in the number of data points used to represent the mixture distribution CDF.

4.3 Sample Prefetching

Another limitation of the exact weights algorithm is that it is sequential. Weights must be computed and samples must be collected for a single table at a time, with no parallelism. To address this shortcoming, Plexus attempts to speculatively collect samples in parallel before they are required in the sample collection phase. This reduces the time spent performing sample collection since any samples that have been prefectched can be used directly without being fetched from the remote sites.

Algorithm 2: Network Traffic Optimization

```
θ ← latency-optimal allocation
if θ satisfies eq. 3 then
    return θ
end

θ ← true mixture allocation
while Time Remaining do
    s1, s2 ← random sites s.t. s1 ≠ s2
    s_fast, s_slow ← The lower and higher latency site
      respectively between s1 and s2
    α ← max change in allocation from s_slow to s_fast
      which satisfies eq. 3. Computed directly by solving eq.
      4.
    θ[s_slow] ← θ[s_slow] − α
    θ[s_fast] ← θ[s_fast] + α
end
return θ
```

In theory, we could continuously prefetch samples for a given table as soon as we have finished generating the table's weights. However, each site may have multiple tables and we need to avoid penalizing other weight generation or sampling jobs. For this reason, we incorporate prefetching as part of a larger pipeline, which ensures that two network (or CPU) intensive operations are not executing concurrently at a single site. We introduce an orchestrator task, which is responsible for aggregating results from each site and producing the final join result. Since sample prefetching is

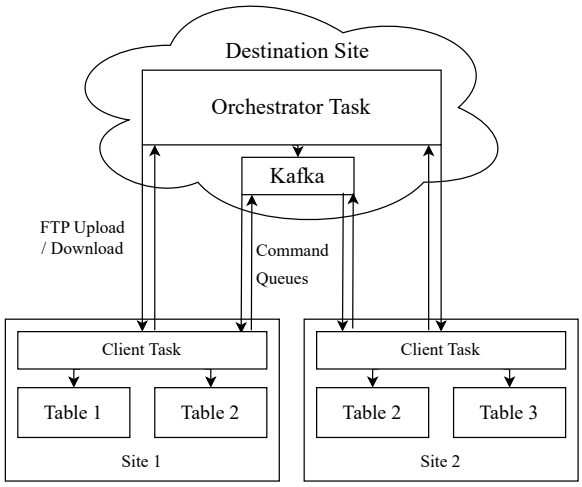


Figure 5: Overview of Plexus Implementation

a network-intensive operation, it is performed while CPU intensive operations are being executed at each site.

Figure 4 illustrates one instantiation of this pipeline, using the same three tables from the example in section 3, partitioned across two sites. Weights are generated for T_3 first, following the steps specified in algorithm 1. Next, weights for T_2 are generated and samples for T_3 are prefetched *in parallel*, since weight generation is compute intensive and prefetching is network intensive. Once the weights for T_2 are generated, prefetching is stopped and the weights are transferred to the orchestrator. This parallel process is repeated for T_2 and T_1, which completes the weight generation phase. The only parallel operation that occurs during sample collection is the orchestrator solving the network optimization problem (algorithm 2) for the next table while samples are being collected from the current table.

Sample prefetching is not guaranteed to produce samples that will participate in the join, which is a limitation of this approach. When samples are prefetched, we have the local table weights available, but it is unknown if any of those tuples join with the next table. We find that it works well in practice (section 5.4), but it is not guaranteed. Our prefetched samples serve a purpose, even if they do not participate in the join. First, the data transfer can be used to estimate network conditions, which is required to take advantage of network heterogeneity. Secondly, the samples themselves can be used to measure data similarity across sites, which may be used to guide the degree to which we exploit network heterogeneity.

4.4 Implementation

Our implementation of Plexus consists of two components: an orchestrator task and a client task, which can be run on a single machine or distributed across many. Communication between tasks is performed using an Apache Kafka instance, which we always run on the same machine as the orchestrator. The orchestrator uses Kafka to send commands to each site, where most of the computation is performed. The clients use Apache Spark [35] for processing the local data at each site and send acknowledgements to the orchestrator after each task is complete. The client spawns a new thread to process each inbound command, which allows for tasks to be performed in parallel, such as weight generation and sample prefetching. When file transfers are required, data is sent over an ftp connection between the client site and the orchestrator. Figure 5 displays a high-level diagram of our system implementation.

5 EVALUATION

5.1 Experimental Setup

Testbed and Methodology. For our main experiments, we use a real-world deployment consisting of four m5.xlarge instances, each residing in a distinct AWS region (Tokyo, Northern California, Ohio, and Ireland). For experiments that measure the system's sensitivity to various tuning parameters, we use a local cluster consisting of four nodes connected via gigabit ethernet and each node equipped with 24 processors and 64 GB of memory. Each site is emulated with resources comparable to the m5.xlarge instances we use in the geo-distributed setting. We used measurements collected with `iperf3` across the same AWS regions as input to our framework and injected sleep times to emulate the network throughput. Table 1 shows the throughput numbers we obtained in Mbps. Emulation of network throughput allows for more accurate comparisons when query latency is not our evaluation target.

Table 1: Network Bandwidth (in Mbps to N. California)

Northern California	Ohio	Ireland	Tokyo
2160	247	88	109

Datasets and Queries. We use three real-world datasets to evaluate our system:

- *TPC-H*: The TPC-H benchmark [31] with 10 GB of data residing in each location.
- *HiBench*: A search engine benchmark which relates page ranks and websites visited by users [2]. We used 6 GB of data in each location.
- *IMDb*: Relational data for movie titles maintained in the International Movie Database [1]. This dataset has about 2 GB of data in each location.

Our target queries are identified in table 2. The TPC-H queries match those used to evaluate centralized exact weights [38].

Algorithms for comparison. We compare the following distributed join strategies in our evaluation:

- *Shuffle*: The default Spark Join which shuffles the full tables over the WAN.
- *Filtering*: A standard distributed join (sampling) technique which uses bloom filters to avoid shuffling tuples which will not participate in the join (e.g. ApproxJoin [26], BloomJoin [27], and AggFirstJoin [14]). By default, we use bloom filters with a false positive rate of 1%.
- *GDEW*: Our Geo-Distributed Exact Weights algorithm
- *Plexus*: Our proposed system. Unless otherwise stated, Plexus is configured with a weight error bound of $\epsilon = 0.5$ and a performance bias of $\delta = 1.0$. We also evaluate how various selections of ϵ and δ affect query latency and sampling accuracy.

Metrics. We use the following metrics to evaluate the baselines and our proposed algorithms:

- *Query latency*: Our primary metric of interest is the time it takes to execute the query, which is measured in seconds.
- *Network traffic*: Reduced network traffic is desirable to reduce query latency since the network is the main bottleneck in the wide-area.
- *Sampling accuracy*: We also evaluate the accuracy of our generated samples, i.e., the degree to which the key frequencies in our sample align with the true key frequencies in the full join result. We use a Chi-Squared test, which evaluates how well the observed frequencies of a given (discrete) variable align with the theoretical frequencies (which would be the true key frequencies in the join result). The test generates an associated p-value, which traditionally indicates a difference in distribution when the p-value is less than 0.05.
- *Aggregate accuracy*: We compute a few aggregate functions to further evaluate the effect of sampling on the accuracy of common queries involving joins and aggregation. We report the absolute percent error associated with each aggregate.

By default, we randomly partition the available tables such that a subset of every table is present at each site with 50% of the join keys unique to that site and 50% overlapping with other sites. Section 5.5 evaluates the effects of different partitioning scenarios.

5.2 Approximate Join Queries

For these experiments, we use one distributed site in each of the following AWS regions: Northern California, Ohio, Japan, and Ireland. We also place the orchestrator in the Northern California region, meaning we will require the resulting samples to be transferred to that location.

Figures 6a and 6b show the query latency across our four algorithms with the TPC-H data. For Q-1, we observe that both geo-distributed algorithms readily outperform the baselines. With a sample size of 1 million tuples, filtering outperforms the full shuffle by 7% while the GDEW algorithm reduces the query latency by 76%. Plexus makes further improvements, reducing latency by approximately 19% over GDEW. We see a similar pattern for Q-2 in figure 6b, with improvements over the full shuffle baseline of 7%, 74% and 81% for filtering, GDEW, and Plexus respectively at the 1 million sample size. However, the query latency increases faster compared to Q-1, since there are more tables in the join and the result is substantially larger than the size of any of the tables.

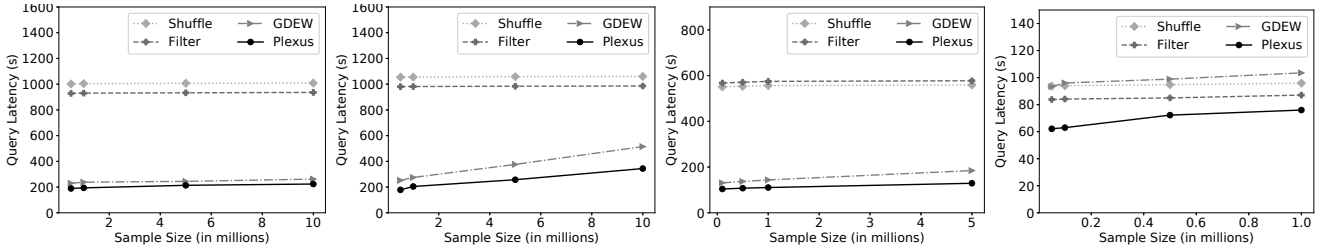
Figure 6c shows the query latency across our four algorithms with Q-3 on the HiBench data. We observe substantially lower latency when using the geo-distributed approaches: a 75% and 80% reduction for GDEW and Plexus respectively for 1 million samples. We also note that the filtering method was more expensive than the full shuffle in this case. This can happen when every key is present in the join result. In that case, the filtering introduces additional overhead without reducing the overall query latency.

Figure 6d shows the query latency across our four algorithms with Q-4 using the IMDb data. This dataset is comparatively small with only about 4GB of data (2GB per location), so we only distribute this across two sites: one in California and the other in Ohio. We observe that both the full shuffle and filtering approaches outperform GDEW across most sample sizes. In this case, it is faster to simply shuffle data over the network rather than attempt weight generation with no optimizations. Plexus system outperforms the existing approaches across all sample sizes, due to its additional optimizations, but the improvement is smaller compared to the other queries and datasets (about 20%). We conclude that there is less benefit to running a geo-distributed algorithm when the data is only a few gigabytes in size.

Network Traffic. Since the network is the main bottleneck in the geo-distributed setting, we briefly examine the amount of data sent over the network by each method. Figure 7 shows the normalized network traffic (compared to the shuffle baseline) generated by each method. The GDEW and Plexus approaches generated much less traffic over the WAN, which accounts for the large performance improvements. Plexus also generated 13%, 38%, 37% and 30% less traffic compared to GDEW on TPC-H (Q-1), TPC-H (Q-2), HiBench, and IMDb respectively. Plexus further reduces traffic by exploiting network heterogeneity and weight approximation.

Table 2: Target Queries

ID	Dataset	Full Join Size	Description	Query
Q-1	TPC-H	4 billion tuples	A three table join on the *customer*, *orders*, and *lineitem* tables.	`SELECT * FROM customer, orders, lineitem WHERE c_custkey = o_custkey AND l_orderkey = o_orderkey;`
Q-2	TPC-H	4 trillion tuples	A five table join on the *nation*, *supplier*, *customer*, *orders*, and *lineitem* tables.	`SELECT * FROM nation, supplier, customer, orders, lineitem WHERE n_nationkey = s_nationkey AND s_nationkey = c_nationkey AND c_custkey = o_custkey AND o_orderkey = l_orderkey;`
Q-3	HiBench	235 million tuples	A join on *UserVisits* and *PageRank*.	`SELECT * FROM UserVisits, PageRank WHERE uv_destURL = pr_pageURL;`
Q-4	IMDb	80 million tuples	A two-table join on title *akas* and *basics*.	`SELECT * FROM akas, basics WHERE akas_titleId = basics_tconst;`



(a) Q-1, TPC-H Data **(b) Q-2, TPC-H Data** **(c) Q-3, HiBench Data** **(d) Q-4, IMDb Data**

Figure 6: Query latency as a function of sample size for a variety of approximate joins.

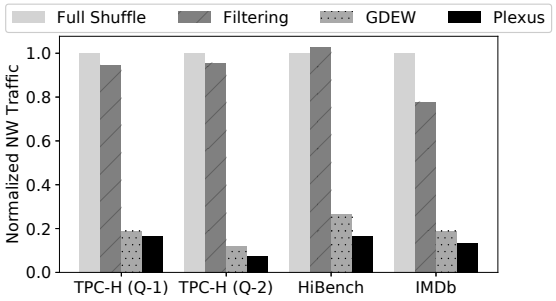


Figure 7: Network traffic generated by each method.

Sampling Accuracy. Introducing optimizations that approximate weights and leverage network heterogeneity could potentially change the key distribution in the join sample. We now evaluate the degree to which our sampled keys are consistent with the true key distribution in the join result. Table 3 shows the results of these experiments using a Chi-Squared test to evaluate deviation. In all cases, we observe very high p-values associated with each comparison, which indicates an extremely high degree of consistency between our sample distribution and the true distribution in the full join. The GDEW algorithm always produces samples that are consistent with the true key distribution. This is expected, since it computes the exact weight for each tuple and does not rely on any kind of approximation. We observe slightly smaller p-values for Plexus, which is unsurprising given the approximation involved. However, the generated samples are also extremely consistent with the true key distribution (for our default parameters ϵ and δ) and nowhere near the traditional value of 0.05, which could be used to detect a statistically significant difference.

Table 3: Join Sampling p-values for a Chi-Squared Goodness of Fit Test

	TPC-H (Q-1)	TPC-H (Q-2)	HiBench	IMDb
GDEW	0.999	0.999	0.999	0.999
Plexus	0.997	0.994	0.986	0.983

5.3 Aggregate Queries

Our join approximation techniques produced accurate samples, i.e. the distribution of keys in the full join did not differ significantly from the key distribution in our sample. We now evaluate the accuracy of aggregate queries that depend on other columns in the sample. For all aggregates, we run the query 10 times and report the average error, along with a 95% confidence interval. We also evaluate the accuracy of the default sampling mechanism in Spark, which we use as a baseline.

In figures 8a and 8b, we evaluate the accuracy of an AVG query and a query for the 90th percentile respectively. In both cases, we are searching for titles that are type "movie" and acquiring information about their run times. We observe that the GDEW approach and sampling with Spark produce very similar results. We observe a small increase in average error with Plexus, but it is not statistically significant (since the confidence intervals are overlapping).

In figures 8c and 8d, we evaluate the same AVG and 90th percentile queries using the revenue generated by each webpage in the HiBench dataset. For both the AVG query and the 90th percentile query, we observe a small increase in average error with Plexus compared to the baselines, but in most cases the difference is not statistically significant. Overall, Plexus produces aggregates that are very close to what we can expect with centralized sampling.

5.4 Optimization Effects

Each of our optimization techniques contributed toward a reduction in latency. We now evaluate the contributions of each approach to the overall reduction in query latency. The sample size was 1 million for these experiments.

Figure 9 shows the effects of each technique on query latency. We consistently observe that sample prefetching has the strongest effect. Heterogeneity awareness and weight approximation have smaller effects in the IMDb experiment, since the data size is relatively small. Note that the benefits of our optimizations are not necessarily additive; sample prefetching reduces the benefits of heterogeneity awareness by reducing the number of samples required. However, the benefits of weight approximation are largely independent of the other approaches. Weight approximation also reduces the amount of computation required at the orchestrator and other sites by eliminating some keys from consideration.

5.5 Sensitivity Analysis

Weight Approximation. Our first optimization reduced the amount of network traffic by approximating weights rather then sending all of them. We introduced a weight error bound ϵ which controls the degree to which error can be introduced in our weight estimation. It represents the average error allowed for weight approximation, which was set to 0.5 for the main experiments. To evaluate its more general effects, we use the IMDb dataset and evaluate how different values of ϵ impact query latency and sample uniformity.

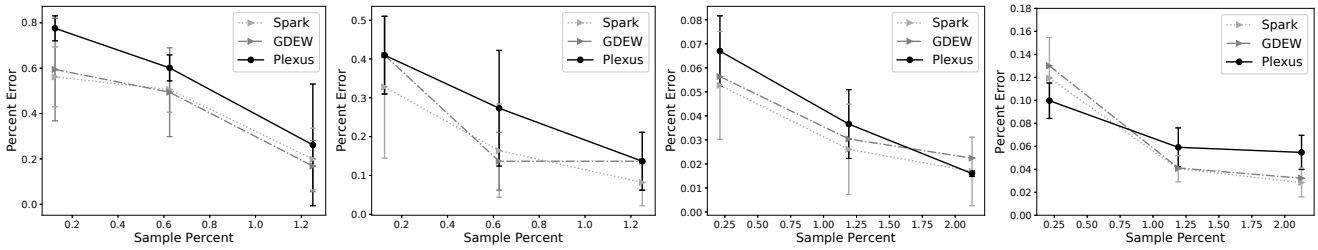
Figure 10a shows the results for the latency experiments. We observe that query latency decreases for increasing values of ϵ. This is expected, since a high tolerance for error will allow us to send fewer weights and spend less compute time gathering samples for keys that were excluded. The dashed line indicates the time taken for the GDEW algorithm and all positive values of ϵ outperform the baseline latency as expected. However, too much approximation error will result in non-uniform samples from the join result.

Figure 10b shows how the generated sample keys diverge from the true key distribution with varying values of ϵ. A Chi-Squared test was used to generate the associated p-values. The distribution of sampled keys diverges as ϵ increases, since we are sampling fewer and fewer weights in each case. We observe a sharp decrease in quality as ϵ gets larger, but ϵ values less than 1.0 generate samples statistically consistent with the true key distribution.

Performance Bias. Our main experiments did not consider distributions of non-joining columns, only key distributions. We now examine how the selection of a performance bias δ affects query latency and aggregate accuracy.

Figure 10c shows the results for varying values of δ. The dashed line indicates the query latency for GDEW, where no network heterogeneity is considered. We observe that the query latency decreases as our preference for a performance bias increases. This is expected, since we can have a higher preference for faster sites when δ is large. This performance gain comes at a cost, since the resulting distribution in the value column can change (the maximum deviation between CDFs is allowed to be δ). Figure 10d shows the performance of the IMDb AVG query with varying values of δ and a sample size of 1 million. We compute the query 10 times for each value of δ and report the mean and a 95% confidence interval. We observe that the average error increases as δ increases, which is expected since we are allowing more data to be collected from the lowest latency edge. However, we do not observe a statistically significant difference across 10 runs. In practice, the selection of δ depends on the application and its tolerance for bias in the data distribution.

Data Placement. We now examine how data placement affects query latency using the IMDb data. We evaluate (1) the degree of cross-site dependency by varying the amount of shared join keys and (2) tables that are not partitioned, but are unique to a given site. Figure 11a shows how different partitioning strategies affect query latency. No key overlap (0%) indicates that sites have no join keys in common, whereas 100% indicates that every join key is present at both sites. As the percentage of shared keys increases, we see the latency decreases. More shared keys allows for parallelism in the sampling phase and the ability to leverage heterogeneity awareness in the case of Plexus. Figure 11b shows the query latency when tables are unique to each site rather than partitioned. There is a performance penalty for both GDEW and Plexus in this case, since stragglers are now required to send the entire sample for their local table, rather



(a) AVG Query, IMDb (b) 90th Percentile, IMDb (c) AVG Query, HiBench (d) 90th Percentile, HiBench

Figure 8: Aggregate query accuracy over various approximate join algorithms.

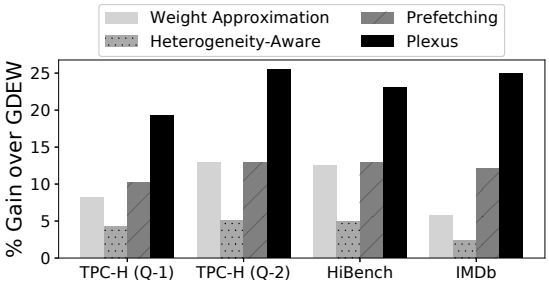


Figure 9: Individual optimization effects on overall query latency compared to GDEW.

than having samples collected from multiple sites in parallel.

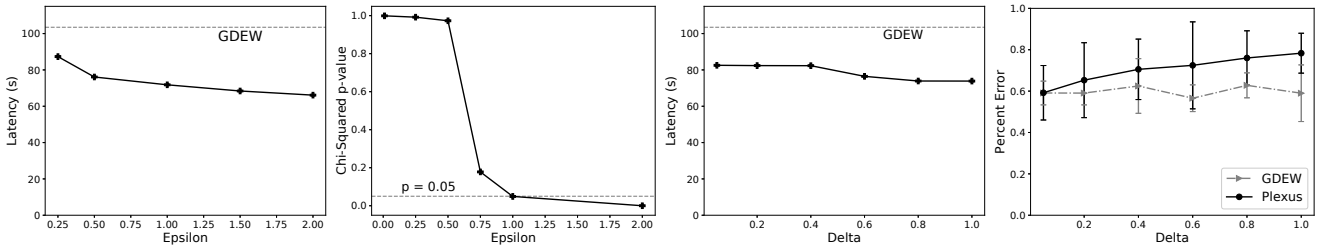
Number of Sites and Network Heterogeneity. We now explore how the number of sites and the degree of network heterogeneity impacts query latency. For some applications, there may be a very large number of sites with relatively stable WAN connections (e.g. all sites exist in a specific country). Alternatively, there could be a small number of sites, but they are distributed across different countries with large degrees of network heterogeneity. We evaluate how Plexus performs in each of these situations, using the IMDb data.

Figure 12 shows how each system topology affects query latency. In Figure 12a, we observe that, all else held constant, an increase in the number of (homogeneous) sites results in a reduction in query latency. This is due to the fact that we are using a fixed sample size; when the number of sites grows, the samples can be collected in parallel from each site. We also observe that as the number of sites continues to increase, the benefits level out. After enough sites are added, the sample size becomes sufficiently small, such that further parallelism provides little to no benefit.

Figure 12b examines the behavior of our geo-distributed algorithms when the variability in network bandwidth increases. In these experiments, we use a mean of 300 Mbps and slowly increase the standard deviation (SD), so there is more variability in the network throughput across sites. Our four sites are assigned a bandwidth of 1 and 2 SDs below the mean and 1 and 2 SDs above the mean. For example, for a SD of 10, our four connections would be 280, 290, 310, and 320 Mbps. We first note that the GDEW latency increases as the network heterogeneity increases. This is due to the fact that GDEW must collect weights from the slowest connection to maintain perfect sample uniformity. For Plexus, we observe that an increase in network heterogeneity *does not* correlate with an increase in latency (for $\delta = 1.0$). In these cases, Plexus is able to leverage low-latency sites to mitigate the effects of slower connections. The query time actually decreases as faster connections become available.

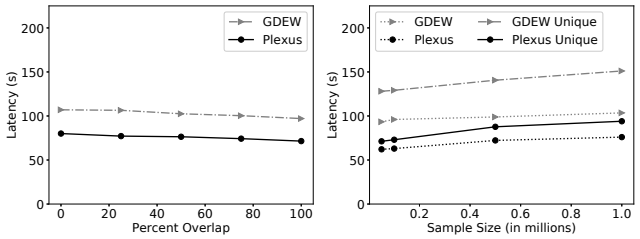
Data Compression. Intermediate results and full tables can be compressed before they are shuffled over the network. We perform an experiment enabling compression with full shuffle, GDEW, and Plexus and compare the resulting latencies. We use the Snappy compression algorithm to compress all data prior to sending.

Figure 13 shows the results for these experiments on the TPC-H data. We observe that the performance of all three techniques are closer when compression is enabled. The geo-distributed algorithms benefit less from compression, since they already attempt to minimize the amount of network traffic. The full shuffle approach has the most to gain, since it is the most network intensive. We conclude that compression benefits all techniques, but Plexus still provides a performance improvement in this setting of around 61% over the full shuffle approach at the 1 million sample size. Compression benefits the full shuffle technique the most, with a reduction in latency around 50%, while the reduction in latency for GDEW and Plexus was 10% and 4% respectively. When both of our proposed methods enable compression, we observe that Plexus still outperforms GDEW by 5% for Q-1 and 13% for Q-2 at the 1 million sample size. Larger sample sizes will result in additional performance improvements for Plexus over GDEW, since Plexus can perform prefetching and leverage network heterogeneity. We also note that for



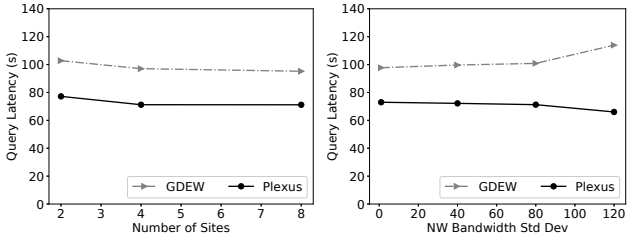
(a) ϵ effect on latency **(b) ϵ effect on accuracy** **(c) δ effect on latency** **(d) δ effect on AVG accuracy**

Figure 10: Evaluation of the tuning parameters in Plexus



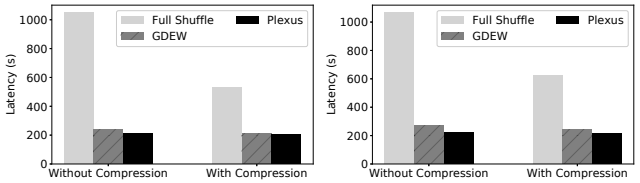
(a) Degree of Overlap **(b) Tables unique to each site**

Figure 11: Effect of Data Placement



(a) Number of Sites **(b) Network Heterogeneity**

Figure 12: Effect of System Heterogeneity



(a) Q-1 **(b) Q-2**

Figure 13: Latency comparison with compression

both queries, our Plexus baseline without compression is able to outperform all other methods, even when they have compression enabled.

5.6 Summary of Results

Our geo-distributed exact weights algorithm showed consistent improvement over the baselines *for sufficiently large datasets*. Even with a conservative choice for ϵ, Plexus resulted in a notable further reduction in query latency. Plexus and GDEW generated substantially less WAN traffic, while Plexus also exploited network heterogeneity to gain additional latency benefits. These results held across a variety of queries, data distributions, and network configurations. We found that sample prefetching offered the largest benefit in terms of latency reduction, followed by weight approximation and heterogeneity awareness. The degree to which non-uniformity can be introduced also affects the latency. If an application can tolerate larger values for ϵ and δ, additional performance improvements can be obtained.

Our proposed algorithms are more performant in many geo-distributed deployments, but in certain settings, shuffling or filtering the data over the network may be preferable. The baselines can be useful when the data size is small or when the network connections are very fast. In these cases, the overhead of generating weights and collecting samples may provide little or no benefit. However, we found that in a real-world deployment with datasets more than a few gigabytes in size, the geo-distributed algorithms offer a substantial reduction in query latency.

5.7 Discussion

Optimizations for selection predicates were not evaluated directly, but can easily be implemented as part of Plexus. Distributed query plans typically need to consider when to apply a given predicate. In the geo-distributed setting, the WAN is a substantial bottleneck, so we recommend pushing these filters to each site, which would reduce network traffic.

Additional optimizations could be built into Plexus that make improvements for recurring queries or incremental approximation. If a given query may be executed multiple times, there are opportunities to reduce the cost on subsequent executions. If the data is static, it suffices to simply

cache the weights from the first execution and perform additional sample collection phases. If the query recurs over dynamically changing data, the weights for each tuple may need to be updated. In this case, the weight generation phase will only need to generate new weights for the tuples that changed, which reduces network traffic and therefore query latency. Alternatively, join synopses could be maintained for each table that would make updates more efficient [39]. Note that the full shuffle and filtering methods would need to send any newly obtained data in the recurring query case.

In many real-world scenarios, data encryption would be required when transferring data between locations. While our implementation used FTP for data transfer, encrypted protocols and/or tunneling may be necessary in practice. This will only increase the importance of efficient use of the network, since encryption keys must be established and encrypting all traffic will drive an increase in I/O latency.

6 RELATED WORK

Geo-Distributed Data Analytics. Performing various analytical tasks in the wide-area is an important related area of research, due to the network scarcity and heterogeneity [20]. General analytics systems like Iridium and WANalytics propose optimizing the data placement and queries to minimize network usage or query latency [24, 33]. MapReduce jobs have also been optimized for the geo-distributed setting by considering network and compute heterogeneity across nodes [9]. These geo-distributed techniques offer important performance improvements, but still generate large shuffles (linear in the table sizes) when computing joins, similar to Apache Spark [35] in our evaluation. Other systems such as PGPregel focus on optimizing analytics over geo-distributed graphs, which is challenging due to the high cost associated with data transfers [40]. Streaming analytics in the wide-area has also received substantial attention [8, 13, 34, 36]. Systems such as AWStream make dynamic adjustments to account for the variation in WAN performance when streaming data. Databases such as CockroachDB have been architected to address the challenges of storing and querying data distributed across the globe [30]. Other relevant lines of research include techniques for scheduling jobs to take advantage of data locality [12] and optimizing geo-distributed data transfers based on an awareness of live network conditions [19]. We devise an algorithm and optimizations to significantly advance the state-of-the-art for join approximation in geo-distributed environments. Furthermore, we leverage data-awareness to constrain optimizations that may affect query accuracy.

Join Approximation. Plexus generalizes the exact weights algorithm suggested by Zhao *et al.* [38], but other variants of the algorithm exist which depend on *rejection sampling* or *online aggregation*. While these techniques could also be implemented in the wide-area, making sequential passes over partitioned tables is more expensive in the geo-distributed setting. Both techniques would require several additional passes over the tables to obtain the desired sample size. However, computing weights for these variants is cheaper, so a future work could consider optimizing these sequential passes to see they are viable in the wide-area.

Distributed join research (within a single datacenter) is another important research direction. AdaptDB [21], Track join [23], and Flow-join [28] are all systems that attempt to reduce join query latency over distributed data, using techniques such as data placement optimization, filtering, network traffic scheduling, and data partitioning. Bloom filter techniques exist for filtering out tuples that will not participate in the join [27], which we used as a baseline in our evaluation. ApproxJoin also uses bloom filters, but is limited to evaluating aggregation functions, not generating tuples from the join result [26]. A common shortcoming of these approaches is that they still must shuffle all tuples participating in the join, since they are designed to produce the full join. More careful weighting of each tuple is required to obtain uniform samples from the join result. AggFirstJoin examines how stream operators can be re-ordered to substantially reduce wide-area network traffic over join queries that compute aggregates [14]. This technique is also restricted to queries that exclusively evaluate an aggregate function. One preliminary work explores factors that influence join sampling latency in the wide-area [15]. It attempts to use the exact weights algorithm; however, it implicitly assumes each table joins on the same key, making it restrictive in practice. Conditional distributions are required to fetch weights in parallel from each site and maintain uniformity [29]. We fully generalize the exact weights algorithm using dynamic programming to support joining on arbitrary columns and introduce three additional optimizations to reduce query latency.

7 CONCLUSION

We presented Plexus, our geo-distributed join approximation system. We generalized the existing Exact Weights algorithm for join sampling and provided three optimizations for reducing query latency: weight approximation, heterogeneity awareness, and sample prefetching. We also explored how minimizing query latency affects other query aspects, such as sample uniformity and bias in non-joining columns. Our analysis shows that our approach can substantially reduce the time required to approximate joins, by up to 80%.

ACKNOWLEDGEMENTS

This research was supported in part by the NSF under grant CNS-1908566.

REFERENCES

[1] [n.d.]. IMDb Non-Commercial Datasets. https://developer.imdb.com/non-commercial-datasets/. Accessed: 2023-02-12.

[2] [n.d.]. Intel HiBench – Big Data Benchmark. https://github.com/Intel-bigdata/HiBench. Accessed: 2023-02-25.

[3] Claude Barthels, Ingo Müller, Timo Schneider, Gustavo Alonso, and Torsten Hoefler. 2017. Distributed Join Algorithms on Thousands of Cores. *Proc. VLDB Endow.* 10, 5 (jan 2017), 517–528.

[4] Matt Calder, Xun Fan, Zi Hu, Ethan Katz-Bassett, John Heidemann, and Ramesh Govindan. 2013. Mapping the Expansion of Google's Serving Infrastructure *(IMC '13)*. Association for Computing Machinery, New York, NY, USA, 313–326.

[5] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. 1999. On Random Sampling over Joins. *SIGMOD Rec.* 28, 2 (jun 1999), 263–274.

[6] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* 55, 1 (2005), 58–75.

[7] Peter J. Haas and Joseph M. Hellerstein. 1999. Ripple Joins for Online Aggregation. *SIGMOD Rec.* 28, 2 (jun 1999), 287–298. https://doi.org/10.1145/304181.304208

[8] Benjamin Heintz, Abhishek Chandra, and Ramesh K. Sitaraman. 2016. Trading Timeliness and Accuracy in Geo-Distributed Streaming Analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (Santa Clara, CA, USA) *(SoCC '16)*. Association for Computing Machinery, New York, NY, USA, 361–373.

[9] Benjamin Heintz, Abhishek Chandra, Ramesh K. Sitaraman, and Jon Weissman. 2016. End-to-End Optimization for Geo-Distributed MapReduce. *IEEE Transactions on Cloud Computing* 4, 3 (2016), 293–306.

[10] Dawei Huang, Dong Young Yoon, Seth Pettie, and Barzan Mozafari. 2020. Joins on Samples: A Theoretical Guide for Practitioners. *Proc. VLDB Endow.* 13, 4 (jan 2020), 547–560.

[11] Yuzhen Huang, Yingjie Shi, Zheng Zhong, Yihui Feng, James Cheng, Jiwei Li, Haochuan Fan, Chao Li, Tao Guan, and Jingren Zhou. 2019. Yugong: Geo-Distributed Data and Job Placement at Scale. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2155–2169.

[12] Chien-Chun Hung, Leana Golubchik, and Minlan Yu. 2015. Scheduling Jobs across Geo-Distributed Datacenters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing* (Kohala Coast, Hawaii) *(SoCC '15)*. Association for Computing Machinery, New York, NY, USA, 111–124.

[13] Albert Jonathan, Abhishek Chandra, and Jon Weissman. 2018. Multi-Query Optimization in Wide-Area Streaming Analytics. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 412–425.

[14] Dhruv Kumar, Sohaib Ahman, Abhishek Chandra, and Ramesh Sitaraman. 2022. AggFirstJoin: Optimizing Geo-Distributed Joins using Aggregation-Based Transformations. In *2023 23rd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*.

[15] Dhruv Kumar, Joel Wolfrath, Abhishek Chandra, and Ramesh K. Sitaraman. 2022. Towards WAN-Aware Join Sampling over Geo-Distributed Data. In *Proceedings of the 5th International Workshop on Edge Systems, Analytics and Networking* (Rennes, France) *(EdgeSys '22)*. Association for Computing Machinery, New York, NY, USA, 13–18.

[16] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *Conference on Innovative Data Systems Research*.

[17] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 615–629.

[18] Mingfeng Lin, Henry C. Lucas, and Galit Shmueli. 2013. Research Commentary - Too Big to Fail: Large Samples and the p-Value Problem. *Inf. Syst. Res.* 24 (2013), 906–917.

[19] Shuhao Liu, Li Chen, and Baochun Li. 2017. Siphon: A High-Performance Substrate for Inter-Datacenter Transfers in Wide-Area Data Analytics *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 646.

[20] Zhengchun Liu, Prasanna Balaprakash, Rajkumar Kettimuthu, and Ian Foster. 2017. Explaining Wide Area Data Transfer Performance *(HPDC '17)*. Association for Computing Machinery, New York, NY, USA, 167–178.

[21] Yi Lu, Anil Shanbhag, Alekh Jindal, and Samuel Madden. 2017. AdaptDB: Adaptive Partitioning for Distributed Joins. *Proc. VLDB Endow.* 10, 5 (jan 2017), 589–600.

[22] Frank Olken. 1993. *Random Sampling from Databases*. Ph.D. dissertation. University of California at Berkeley.

[23] Orestis Polychroniou, Rajkumar Sen, and Kenneth A. Ross. 2014. Track Join: Distributed Joins with Minimal Network Traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 1483–1494.

[24] Qifan Pu et al. 2015. Low Latency Geo-Distributed Data Analytics. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (London, United Kingdom) *(SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 421–434.

[25] Qifan Pu, Ganesh Ananthanarayanan, Peter Bodik, Srikanth Kandula, Aditya Akella, Paramvir Bahl, and Ion Stoica. 2015. Low Latency Geo-Distributed Data Analytics. *SIGCOMM Comput. Commun. Rev.* 45, 4 (aug 2015), 421–434. https://doi.org/10.1145/2829988.2787505

[26] Do Le Quoc, Istemi Ekin Akkus, Pramod Bhatotia, Spyros Blanas, Ruichuan Chen, Christof Fetzer, and Thorsten Strufe. 2018. ApproxJoin: Approximate Distributed Joins. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 426–438.

[27] Sukriti Ramesh, Odysseas Papapetrou, and Wolf Siberski. 2009. Optimizing Distributed Joins with Bloom Filters. In *Distributed Computing and Internet Technology*, Manish Parashar and Sanjeev K. Aggarwal (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 145–156.

[28] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-Join: Adaptive skew handling for distributed joins over high-speed networks. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 1194–1205.

[29] Ali Mohammadi Shanghooshabad, Meghdad Kurmanji, Qingzhi Ma, Michael Shekelyan, Mehrdad Almasi, and Peter Triantafillou. 2021. PGMJoins: Random Join Sampling with Graphical Models. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1610–1622.

[30] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509.

[31] TPC-H Benchmark. Accessed: 2022-11-11. http://www.tpc.org/tpch/.

[32] Raajay Viswanathan, Ganesh Ananthanarayanan, and Aditya Akella. 2016. CLARINET: WAN-Aware Optimization for Analytics Queries. In

12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). USENIX Association, Savannah, GA, 435–450.

[33] Ashish Vulimiri, Carlo Curino, Philip Brighten Godfrey, Thomas Jungblut, Konstantinos Karanasos, Jitendra Padhye, and George Varghese. 2015. WANalytics: Geo-Distributed Analytics for a Data Intensive World. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1087–1092.

[34] Joel Wolfrath and Abhishek Chandra. 2022. Efficient Transmission and Reconstruction of Dependent Data Streams via Edge Sampling. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*. 47–57.

[35] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Commun. ACM* 59, 11 (oct 2016), 56–65.

[36] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. 2018. AWStream: Adaptive Wide-Area Streaming Analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 236–252.

[37] Yikai Zhao, Yinda Zhang, Yuanpeng Li, Yi Zhou, Chunhui Chen, Tong Yang, and Bin Cui. 2022. MinMax Sampling: A Near-Optimal Global Summary for Aggregation in the Wide Area. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 744–758.

[38] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1525–1539.

[39] Zhuoyue Zhao, Feifei Li, and Yuxi Liu. 2020. Efficient Join Synopsis Maintenance for Data Warehouse. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 2027–2042.

[40] Amelie Chi Zhou, Ruibo Qiu, Thomas Lambert, Tristan Allard, Shadi Ibrahim, and Amr El Abbadi. 2022. PGPregel: An End-to-End System for Privacy-Preserving Graph Processing in Geo-Distributed Data Centers. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 386–402.