2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing (CCGrid) | 979-8-3503-0119-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/CCGRID57682.2023.00046

AggFirstJoin: Optimizing Geo-Distributed Joins using Aggregation-Based Transformations

Dhruv Kumar
IIIT Delhi†
New Delhi, India
dhruv.kumar@iiitd.ac.in

Sohaib Ahmad
University of Massachusetts
Amherst, USA
sohaib@cs.umass.edu

Abhishek Chandra
University of Minnesota
Twin Cities, USA
chandra@umn.edu

Ramesh K. Sitaraman
University of Massachusetts
Amherst, USA
ramesh@cs.umass.edu

***Abstract*—Geo-distributed analytics (GDA) involves processing of data stored across geographically distributed sites. Such analytics involves data transfer over the wide area network (WAN) links. WAN links are highly constrained and heterogeneous in nature, making the data transfer over the WAN slow and costly. To tackle this issue, recent approaches have proposed WAN-aware scheduling and placement of geo-distributed analytics tasks. However, computing *joins* in a geo-distributed setting remains a challenging problem. In this work, we propose *AggFirstJoin*, an approach to minimize the cost of geo-distributed joins using a theoretically sound query transformation technique. Our optimization approach takes a combined view of the join and aggregation operations which are often part of the same query and pushes (a transformed) aggregation before join in a manner to produce the same results as the original query. We augment our query transformation technique with a WAN-aware task placement and a Bloom filtering approach to further reduce query execution time and WAN usage respectively. We implement our proposed technique on top of Apache Spark, a popular engine for big data analytics. We extensively evaluate our proposed technique using synthetic, TPC-H and Amplab Big Data benchmark datasets on a real geo-distributed testbed on AWS as well as an emulated testbed. Our evaluations show our proposed technique achieves up to 300x reduction in query execution time and 200x reduction in WAN usage as compared to state-of-the-art GDA techniques.**

I. Introduction

Today, data is generated in a geographically distributed manner in a wide variety of domains such as social networks, e-commerce, search engines, online advertisements, audio and video streaming, energy, smart cities, IoT sensors etc. Consequently, this data is stored across geographically distributed edges and data centers (DCs) near to the end-users and end-devices, the very sources of this data. Analyzing this geographically distributed data is challenging primarily due to two reasons: 1) constrained and costly WAN bandwidth links which connect the geo-distributed edges and DCs (henceforth collectively called as sites) [1], and 2) limited compute availability at each site (especially the edges) [2].

Limitations of state-of-the-art approaches. Prior work has sh-own that transferring all the data to a centralized DC for analysis significantly increases the query execution time as well as the monetary cost of data transfer [3], [4], [5], [2], [6], [7], [8]. Consequently, various approaches have been proposed to perform analytics in a geographically distributed manner. This involves utilizing multiple geographically distributed sites for computing the analytical results. The existing approaches have tried to address the WAN bandwidth and compute capacity constraints by proposing query optimization [1], task placement and scheduling strategies [3], [5], [9], [2], [4] which systematically give preference to those *reduce* sites having higher bandwidth capacity links and/or higher compute capacity. Although these techniques offer a great improvement over centralized query execution, the amount of data shuffled and query execution time for operations such as *joins* can be very high and often remains the bottleneck. This is because even with the existing geo-distributed approaches, joins often involve shuffling of the entire joining tables across the geo-distributed sites and generating the cross product between the joining tables.

Our approach and key insights. In this work, we focus on optimizing the execution of *geo-distributed joins*. We use the insight that in typical queries, the *join* operators are often followed by *aggregation* operators. Our proposed approach optimizes the join by identifying opportunities to push the (transformed) aggregation *before* the join operator in a manner that does not change the final query results (i.e. without any accuracy loss). This query transformation is particularly important in a geo-distributed setting where individual tables may be partitioned across multiple sites. Pushing the aggregation before join can allow us to aggregate table partitions in-situ, and then shuffle only the aggregated partition results across the WAN for computing the final join. Such an approach can save significantly on WAN usage and query execution latency, especially for *multi-way joins* where the aggregation is performed over columns spanning multiple tables.

Challenges. It is challenging to transform a query by pushing aggregation before join correctly and efficiently in a geo-distributed setting. The existing work on pushing aggregation before join [10], [11] supports only single-server databases where all tables are stored in one site and there is only one partition per table. It does not provide answers to the many additional questions posed by geo-distributed joins: (1) Given a query, how to derive a transformation in a principled manner which would work well in a geo-distributed setting? (2) Should the aggregation be performed at one site or across multiple sites in a geo-distributed manner? (3) How to com-

†This work was done while the author was a graduate student at the University of Minnesota, Twin Cities.

979-8-3503-0119-9/23/$31.00 ©2023 IEEE
DOI 10.1109/CCGrid57682.2023.00046

bine the multiple partitions of each table distributed across geo-distributed sites? (4) How to address WAN constraints and heterogeneity in the context of geo-distributed joins? (5) How to avoid sending redundant data over the WAN? Moreover, the existing work does not answer: (6) How to perform aggregation over derived columns which may be derived from columns spanning multiple tables? (7) How to transform higher order aggregation functions (e.g. variance, skewness, kurtosis)? The existing work supports only a limited set including sum, average, min, max and count. (8) How to perform aggregation over columns with data types other than real numbers (e.g. strings)? We present a general framework to push such sophisticated aggregations before the join in a geo-distributed environment.

Research contributions. In this paper, we present AggFirstJoin, a theoretically sound query transformation technique. We identify the operations and query transformations needed to generate an equivalent query execution (resulting in the same result as the original query). We augment our query transformation technique with a WAN-aware task placement method that reduces the query execution latency by exploiting the WAN heterogeneity across multiple sites. Further, to reduce the amount of data sent over the WAN, we employ a filtering technique using Bloom filters to preemptively remove redundant keys which would not appear in the final join result. We summarize our main contributions next:

- We propose AggFirstJoin, an approach to minimize the cost of geo-distributed joins using a theoretically sound query transformation technique.
- We augment our query transformation technique with a WAN-aware task placement and a Bloom filtering approach to further reduce query execution time and WAN usage respectively.
- We implement AggFirstJoin on top of Apache Spark, a popular analytics engine.
- We evaluate our approach using synthetic traces and popular benchmarks on AWS as well as an emulated testbed to show up to 300x reduction in query execution time and 200x reduction in WAN usage as compared to state-of-the-art GDA techniques.

II. Background and Preliminaries

System model. We consider a geo-distributed analytics system [3], [4], [2], [12] spanning across multiple geo-distributed sites. These sites may vary in terms of compute and storage capacity and are connected to each other via wide-area network (WAN) links. A site may just be an edge cluster located closer to a group of user devices but having very limited compute and storage capacity or a site may be a full fledged data center (DC) having abundant compute and storage capacity. Each site continuously ingests data streams from multiple data sources such as user devices and IoT sensors, and stores it locally for batch analysis. It then sends the processed results to a central DC which combines results from all the sites and saves the final result for consumption by analysts.

Processing model. We consider batch processing model which involves running analytics queries over batches of data distributed across geo-distributed sites. For example, a recurring query may be issued every 12 hours for analyzing the user session logs of a social networking service for last 12 hours.

Resource constraints and heterogeneity in GDA. Traditionally, analytics are run in an intra-DC environment which have abundant and homogeneous network and compute resources. On the contrary, geo-distributed analytics involves data transfer over WAN links which have (1) *highly constrained bandwidth*: inter-DC WAN bandwidth is 1-2 orders of magnitude less than intra-DC bandwidth [1], and (2) *heterogeneous bandwidth*: there is a great deal of variation in the bandwidth availability on different WAN links. For example, in a geo-distributed setup on AWS EC2 , the ratio of highest to lowest WAN bandwidth capacity can be $>$ 20 [1]. Additionally, geo-distributed sites can span across edge clusters (constrained compute resources) and full-fledged DCs (abundant compute resources) leading to heterogeneity in compute resources available at each site [2]. Hence, it is important to design solutions for geo-distributed analytics which factor in constraints and heterogeneity in bandwidth and compute availability.

Target queries. We focus on analytics queries that compute joins over multiple tables followed by aggregation over one or more columns. Joins are one of the most compute and network intensive operations and hence, optimizing them is an important problem. We discuss a few examples to further motivate the problem.

- **Example** 1**.** Using the TPC-H benchmark schema [13], a part supplier wants to compute his total profit from the selling of its line items. The SQL statement for such a query would look like:

```
SELECT PS.SuppKey, SUM(LI.Price * (1-LI.Discount)
- LI.Quantity * PS.SuppCost) as Profit
FROM PartSupplier PS, LineItem LI
WHERE PS.SuppKey = LI.SuppKey and PS.PartKey =
LI.PartKey GROUP BY PS.SuppKey
```

- **Example** 2**.** A multinational e-commerce enterprise selling products in multiple countries wants to find the total revenue coming from each of its products. This requires joining over the order table (comprising columns such as order id, product id, currency, amount) and the currency conversion table (comprising columns such as currency, conversion rate). The SQL statement for such a query would look like:

```
SELECT PId, SUM(Amount * Conversion Rate) as
Revenue FROM ORDERS, CURRENCY
WHERE ORDERS.Currency == CURRENCY.Currency
GROUP BY PId
```

Note that all the above examples involve *aggregations over new columns derived from columns spanning multiple tables*.

Metrics. We focus on the following metrics:

- **Latency.** This refers to the total time taken by a query to execute from the start to the finish of query execution. This is important since the results of the analytics queries are often required for taking business-critical decisions. This

latency can be further divided into two major parts: (1) **Network latency**: This is the time taken to transfer data (both input and intermediate) across geo-distributed sites. (2) **Compute latency**: This refers to the time spent in computing the query results (excluding the network latency).
- **WAN usage.** This refers to the actual amount of data transferred across the WAN links. This is important since the data transfer over WAN links is costly [4] unlike intra-DC data transfer which is generally free.

III. Challenges in Geo-Distributed Joins

We identify the challenges in geo-distributed joins and discuss prior work with a simple example. Suppose there are two database tables $T_1(Key, C_1)$ and $T_2(Key, C_2)$ geo-distributed across sites S_1 and S_2. Each site stores portions of T_1 and T_2 each. Let us also assume that T_1 has total R_1 records with each key k having $R_{1,k}$ records. Similarly, T_2 has total R_2 records with each key k having $R_{2,k}$ records. Let us now consider the following join query:

SELECT Key**, SUM(**$C_1 * C_2$**) as** C_{agg} **FROM** T_1, T_2
WHERE $T_1.Key == T_2.Key$ **GROUP BY** Key

The final query results are required to be available at a third site S_3. All the three sites are connected via WAN links. To execute this query, a centralized approach would first transfer the portions of T_1 and T_2 from site 1 and site 2 to site 3, combine the partitions of each table to form one partition per table and then proceed with the join computations over T_1 and T_2. Prior work on GDA such as Iridium [3] and Clarinet [1] would first shuffle the records of T_1 and/or T_2 among S_1 and S_2 to get all the data for each key into a single site and then proceed with the join computations. In both the centralized as well as WAN-aware approaches, the amount of data shuffled across WAN links is proportional to $(R_1 + R_2)$. For a multi-way join over N tables ($N > 2$), the amount of data shuffled would be proportional to $\sum_{i=1}^{N}(R_i)$. Since WAN links are bandwidth constrained as well as costly (in terms of monetary cost), *such data transfers over WAN links are generally very hurtful in terms of both latency and cost*.

Furthermore, a naive execution of the actual join computations would first generate a Cartesian product from the rows in C_1 and C_2. Then, a new intermediate column would be derived from the existing C_1 and C_2 columns. Once C_{new}[1] is created, the joined table would be grouped by Key and aggregated over C_{new} to get the final result C_{agg}. An important point to note here is that the size of the intermediate joined table may increase non-linearly with the size of the tables participating in the join. In this example, the number of rows in the intermediate joined table would be $\sum_k(R_{1,k} \cdot R_{2,k})$. For a multi-way join over N tables ($N > 2$), the size of the intermediate table would be $\sum_k(\prod_{i=1}^{N} R_{i,k})$. Hence, in the case of large tables, the intermediate Cartesian product can blow up in size easily [14]. This can cause even in-memory processing systems such as Apache Spark [15] to spill the intermediate results to disk if the intermediate results cannot be accommodated within the memory. As a result, *the compute latency associated with such joins can also be very high*.

[1]**Note:** C_{new} is derived from existing columns coming more than one table.

Challenges. In conclusion, there are three main challenges in computing geo-distributed joins:
- **High network latency:** Shuffling the raw tables over WAN links often takes a large duration of time because WAN bandwidths are highly constrained.
- **High WAN cost:** Since the shuffled tables over WAN links can be very large in size, such shuffles also lead to high WAN costs.
- **High compute latency:** The size of the intermediate joined table can be orders of magnitude larger than the sum of the sizes of the joining tables. and hence, can lead to high compute latency.

IV. Geo-distributed Joins

A. Recomposable and Decomposable Functions

We first define the mathematical notion of recomposable and decomposable functions, which we use for our optimizations.

Recomposable Functions. Let Σ be the set of possible values. Now consider two sets $A, B \subseteq \Sigma$ of size m and n respectively, a transformation function $\oplus : \Sigma \times \Sigma \to \Sigma$ such that

$$A \times B = \{(a_i, b_j)\} \qquad \forall 1 \le i \le m, 1 \le j \le n$$
$$\oplus(A \times B) = \{a_i \oplus b_j\} \qquad \forall 1 \le i \le m, 1 \le j \le n$$

We define an aggregation function $agg : 2^{\Sigma} \to \Sigma$, i.e. it takes a set of values and maps it to a single value. If there exists aggregation functions, $\mathrm{agg}_1 : 2^{\Sigma} \to \Sigma^l$ and $\mathrm{agg}_2 : 2^{\Sigma} \to \Sigma^l$ and a transformation function $\odot : \Sigma^l \times \Sigma^l \to \Sigma$ such that

$$\mathrm{agg}(\oplus(A \times B)) = \odot(\mathrm{agg}_1(A), \mathrm{agg}_2(B)) \tag{1}$$

then the pair $(\mathrm{agg}, \oplus)$ is called as a *recomposable function pair*, agg_1 and agg_2 are called *recomposed aggregation functions*, and $\odot$ is called a *recomposed transformation*.

Recomposable function pairs cover an extensive set of aggregations and transformations. For instance, common aggregation functions such as SUM, COUNT, AVG, MIN, MAX, VAR etc. paired with any transformation expression comprising addition ($+$), subtraction ($-$), multiplication ($\cdot$), and division($\div$) are covered under this property. Table I lists some of these function pairs $(\mathrm{agg}, \oplus)$ and their corresponding $(\odot, \mathrm{agg}_1,\ \mathrm{agg}_2)$ functions. In general, this recomposability property can be applied to more general data types such as strings and can also be combined with local map transformations such as mapping a string to a numeric value.

Example 1. Let us consider the function pair $(\mathrm{SUM}, +)$ and $A = \{2, 3, 4\}, B = \{5, 6\}$

$$A \times B = \{(2,5), (2,6), (3,5), (3,6), (4,5), (4,6)\}$$
$$\mathrm{SUM}(+(A \times B)) = (2+5) + (2+6) + (3+5) + (3+6) + (4+5) + (4+6) = 51$$

$\mathrm{SUM}(+(A \times B))$ can be recomposed using

$$\begin{aligned}\text{agg}_1(A) &= (\text{SUM}(A), \text{COUNT}(A)) = (9, 3)\\ \text{agg}_2(B) &= (\text{SUM}(B), \text{COUNT}(B)) = (11, 2)\\ \odot(\text{agg}_1(A), \text{agg}_2(B)) &= \text{SUM}(A) \cdot \text{COUNT}(B) + \text{SUM}(B) \cdot \text{COUNT}(A)\\ &= 9 \cdot 2 + 11 \cdot 3 = 51\end{aligned}$$

In this way, we see that

$$\text{SUM}(+(A \times B)) = \text{SUM}(A) \cdot \text{COUNT}(B) + \text{SUM}(B) \cdot \text{COUNT}(A)$$

Example 2. Let us consider the function pair $(\text{LONGEST}, \text{CONCAT})$ and $A = \{\text{Tim}, \text{Jack}\}, B = \{\text{King}, \text{Lee}\}$

$$\begin{aligned}A \times B &= \{(\text{Tim}, \text{King}), (\text{Tim}, \text{Lee}), (\text{Jack}, \text{King}), (\text{Jack}, \text{Lee})\}\\ \text{LONGEST}(\text{CONCAT}(A \times B)) &= \text{LONGEST}\{\text{Tim King}, \text{Tim Lee},\\ &\qquad \text{Jack King}, \text{Jack Lee}\} = \text{Jack King}\end{aligned}$$

$\text{LONGEST}(\text{CONCAT}(A \times B))$ can be recomposed using

$$\begin{aligned}\text{agg}_1(A) &= \text{LONGEST}(A) = \text{Jack}\\ \text{agg}_2(B) &= \text{LONGEST}(B) = \text{King}\\ \odot(\text{agg}_1(A), \text{agg}_2(B)) &= \text{CONCAT}(\text{LONGEST}(A), \text{LONGEST}(B))\\ &= \text{Jack King}\end{aligned}$$

In this way, we see that

$$\text{LONGEST}(\text{CONCAT}(A \times B)) = \text{CONCAT}(\text{LONGEST}(A), \text{LONGEST}(B))$$

Decomposable Functions. Let $A_1, A_2, ..., A_D$ be disjoint subsets of A such that $A_1 \cup A_2 \cup ... \cup A_D = A$. Then if the following holds

$$\text{agg}(A) = \text{agg}(\{\text{agg}(A_1), \text{agg}(A_2), ... \text{agg}(A_D)\})$$

then agg is said to be decomposable. In other words, we can compute the aggregate value (agg) of a set by computing the aggregate value of the aggregate of each disjoint subset. Common aggregation functions such as SUM, COUNT, MIN, MAX etc. satisfy this decomposability property.

B. Aggregation Before Join

We now explain the systematic procedure for performing aggregation before join. We first optimize a join over two tables, each having just a single partition i.e. without any geo-distribution. We then extend it to include geo-distributed partitions. Finally we generalize it to optimize multi-way joins.

Two-way join. Let us consider two tables T_1 and T_2 with schema (K, C_1) and (K, C_2) respectively where K is the join key[2], and C_1 and C_2 are the value columns. Each table has only one partition and is stored at the same site i.e. there is no geo-distribution. The join query over these two tables can be written as:

SELECT K, $agg(\oplus(C_1, C_2))$ **FROM** T_1, T_2
WHERE $T_1.K = T_2.K$ **GROUP BY** K

Here[3], function $\oplus$ is a transformation function which derives a new column $\oplus(C_1, C_2)$ as a function of C_1, C_2 columns (columns are from more than one table) and agg is any aggregation function which is used to aggregate the derived column after grouping by unique values in K.

[2]The join key K need not be the primary key.

[3]Without loss of generality, we omit other common clauses such as HAVING, ORDER BY etc. as our focus is on optimizing join and aggregation.

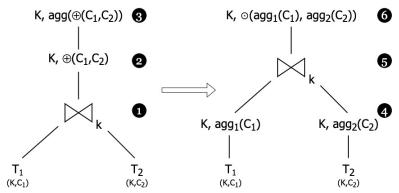


Fig. 1: Two-way Join

NaiveJoin. A typical query execution plan for this query would have the following steps in order: (Also, see Figure 1)

1) Join over T_1 and T_2. This creates a temporary table having attributes (K, C_1, C_2) (❶)
2) Create the new attribute $\oplus(C_1, C_2)$. (❷)
3) Group by K and aggregate $\oplus(C_1, C_2)$ using function agg to get $agg(\oplus(C_1, C_2))$. (❸)

In the above steps, the join operation (Step ❶) is generally the costliest step in terms of both the compute time and data shuffle time. Therefore, our goal is to optimize this step by performing the aggregation (Step ❸) before the join step. Intuitively, we want to aggregate over tables T_1 and T_2 individually, and then perform the join operation over the aggregated tables. After the join operation, an additional transformation would be required in order to get the final aggregated result $agg(\oplus(C_1, C_2))$ which requires information from both tables.

AggFirstJoin. Our modified query execution plan would have the following steps in order: (See Figure 1)

1) For each table, group by K and aggregate C_1 as $agg_1(C_1)$ and C_2 as $agg_2(C_2)$. (❹)
2) Join over aggregated tables $T_1(K, agg_1(C_1))$ and $T_2(K, agg_2(C_2))$. (❺)
3) Derive the new column as $\odot(agg_1(C_1), agg_2(C_2))$. (❻)

Notice that in *NaiveJoin*, function $\oplus$ and agg require the joined table where as in *AggFirstJoin*, functions agg_1 and agg_2 act on individual tables T_1 and T_2 respectively and hence, can be computed before the join. Function $\odot$ acts on the results of the joined table and hence, is computed after the join.

Theorem 1. *Given two tables T_1 and T_2, performing AggFirstJoin on them yields the same result as NaiveJoin.*

Proof. Follows from the recomposability property of $(\text{agg}, \oplus)$.

Distributed Joins. Let each table T_1 and T_2 be distributed across D nodes[4]. T_{1d} and T_{2d} represent the dth partition of T_1 and T_2 respectively. To be able to perform aggregation before the join over distributed tables, agg_1 and agg_2 must be *decomposable* functions.

Figure 2 shows how a distributed join would be optimized. We can first aggregate individual partitions of each table using agg_1 and agg_2 and then combine them to get a single aggregated partition per table. Then we would proceed with

[4]These nodes could be in one DC or geographically distributed across multiple DCs.

$\oplus$	$\text{agg}_1(A)$	$\text{agg}_2(B)$	$\odot(\text{agg}_1(A), \text{agg}_2(B))$
$+$	$(\text{SUM}(A), \text{COUNT}(A))$	$(\text{SUM}(B), \text{COUNT}(B))$	$\text{SUM}(A) \cdot \text{COUNT}(B) + \text{SUM}(B) \cdot \text{COUNT}(A)$
$\cdot$	$\text{SUM}(A)$	$\text{SUM}(B)$	$\text{SUM}(A) \cdot \text{SUM}(B)$

(a) Transformations for SUM

$\oplus$	$\text{agg}_1(A)$	$\text{agg}_2(B)$	$\odot(\text{agg}_1(A), \text{agg}_2(B))$
$+$	$(\text{SUM}(A), \text{COUNT}(A))$	$(\text{SUM}(B), \text{COUNT}(B))$	$\frac{\text{SUM}(A)\cdot\text{COUNT}(B)+\text{SUM}(B)\cdot\text{COUNT}(A)}{\text{COUNT}(A)\cdot\text{COUNT}(B)}$
$\cdot$	$(\text{SUM}(A), \text{COUNT}(A))$	$(\text{SUM}(B), \text{COUNT}(B))$	$\frac{\text{SUM}(A)\cdot\text{SUM}(B)}{\text{COUNT}(A)\cdot\text{COUNT}(B)}$

(b) Transformations for AVG

$\oplus$	$\text{agg}_1(A)$	$\text{agg}_2(B)$	$\odot(\text{agg}_1(A), \text{agg}_2(B))$
$+$	$\text{COUNT}(A)$	$\text{COUNT}(B)$	$\text{COUNT}(A) \cdot \text{COUNT}(B)$
$\cdot$	$\text{COUNT}(A)$	$\text{COUNT}(B)$	$\text{COUNT}(A) \cdot \text{COUNT}(B)$

(c) Transformations for COUNT

$\oplus$	$\text{agg}_1(A)$	$\text{agg}_2(B)$	$\odot(\text{agg}_1(A), \text{agg}_2(B))$
$+$	$\text{MIN}(A)$	$\text{MIN}(B)$	$\text{MIN}(A) + \text{MIN}(B)$
$\cdot$	$(\text{MIN}(A), \text{MAX}(A))$	$(\text{MIN}(B), \text{MAX}(B))$	$\text{MIN}\Big(\text{MIN}(A)\cdot\text{MIN}(B), \text{MIN}(A)\cdot\text{MAX}(B), \text{MAX}(A)\cdot\text{MIN}(B), \text{MAX}(A)\cdot\text{MAX}(B)\Big)$

(d) Transformations for MIN

$\oplus$	$\text{agg}_1(A)$	$\text{agg}_2(B)$	$\odot(\text{agg}_1(A), \text{agg}_2(B))$
$+$	$\text{SUM}(A), \text{SUM}(A^2), \text{COUNT}(A)$	$\text{SUM}(B), \text{SUM}(B^2), \text{COUNT}(B)$	$\frac{\text{SUM}(A^2)\cdot\text{COUNT}(B)+\text{SUM}(B^2)\cdot\text{COUNT}(A)+2\cdot\text{SUM}(A)\cdot\text{SUM}(B)}{\text{COUNT}(A)\cdot\text{COUNT}(B)} - \Big(\frac{\text{SUM}(A)\cdot\text{COUNT}(B)+\text{SUM}(B)\cdot\text{COUNT}(A)}{\text{COUNT}(A)\cdot\text{COUNT}(B)}\Big)^2$
$\cdot$	$\text{SUM}(A), \text{SUM}(A^2), \text{COUNT}(A)$	$\text{SUM}(B), \text{SUM}(B^2), \text{COUNT}(B)$	$\frac{\text{SUM}(A^2)\cdot\text{SUM}(B^2)}{\text{COUNT}(A)\cdot\text{COUNT}(B)} - \Big(\frac{\text{SUM}(A)\cdot\text{SUM}(B)}{\text{COUNT}(A)\cdot\text{COUNT}(B)}\Big)^2$

(e) Transformations for VARIANCE

TABLE I: **Example Recomposable Function Pairs. For $-$ and $\div$, replace B by $-B$ in $+$ and by $1/B$ in $\cdot$ resp.**

Apache Spark SQL	**Supported by AggFirstJoin**
Sum, Count, Average, Min, Max, First, Last, Variance, Correlation, Covariance, Std Dev, Skewness, Kurtosis, Approx. Count Distinct, Approx. Percentile	**Yes**
Count Distinct, Sum Distinct	**No**

TABLE II: **Comparison of aggregation functions in Apache Spark SQL and AggFirstJoin.**

the join and transformation as in the case of single partition tables.

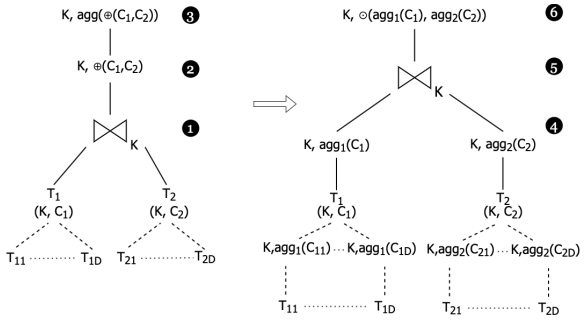


Fig. 2: Two-way Distributed Join

Multi-way Joins. We now extend the above approach to optimize multi-way joins (joins over $q \geq 2$ tables). In this case we would have q aggregation functions $agg_i : 2^{\Sigma} \rightarrow \Sigma^l$ where $1 \leq i \leq q$, $\oplus : \Sigma^q \rightarrow \Sigma$ and $\odot : \{\Sigma^l \times \Sigma^l \times ... \times \Sigma^l\}_q \rightarrow \Sigma$. If $C_i \subseteq \Sigma$ where $1 \leq i \leq q$, then

$$\text{agg}(\oplus(C_1 \times ... \times C_q)) = \odot(agg_1(C_1), ..., agg_q(C_q)) \quad (2)$$

Equation 2 generalizes the recomposability property to an arbitrary number of tables. If the recomposed aggregation functions agg_i are also decomposable, we can perform the multi-way join when the q tables are distributed across different sites.

Enhancements. We now explain how to extend the above approach for arbitrary joins and aggregations having one or more of the below specifications:

- **Multiple aggregation functions in the SELECT clause.** For queries which consist of more than one type of aggregation over same or different columns, *AggFirstJoin* will individually identify the required transformations for each aggregation and combine them to enumerate all the required transformations.
- **Different join and group by attributes.** When the join attributes and the group by attributes are different in a query, then *AggFirstJoin* will perform group by and aggregation over the combination of join and group by attributes before the join operation. Once the join is computed, an additional aggregation over the original group by columns will need to be performed followed by transformation over the joined aggregated results, to yield the same result as *NaiveJoin*.
- **Other types of joins.** *AggFirstJoin* can be easily modified to accommodate other types of joins such as OUTER, LEFT and RIGHT joins. For instance, for LEFT joins, if the COUNT of a key in the left table is non-zero and that in right table is zero, *AggFirstJoin* replaces the zero by one to ensure that all the keys in the LEFT table are included in the joined result.

Combining everything. Algorithm 1 explains AggFirstJoin's query transformation at a high level. For any query consist-

ing of general multi-way joins and arbitrary transformations, *AggFirstJoin* takes the query execution tree built by the query optimizer and tries to push the aggregation operations from the top of the tree down to the leaf of the tree. For any join operation and corresponding aggregation, it identifies the columns coming from each side of the join. It then tries to identify the corresponding recomposed aggregation and transformation functions using the pre-defined recomposition rules (such as those described in Table I). If a suitable recomposition rule is found, it pushes that aggregation before the join operation. It repeats this recursively at each join operation until it reaches the base tables i.e. leaves of the tree.

Generality of AggFirstJoin. Since recomposable and decomposable functions (§IV-A) cover an extensive set of commonly used aggregations and transformations, *AggFirstJoin* can optimize a wide range of join and aggregation queries. Table II shows that *AggFirstJoin* can optimize 15 out of 17 aggregation functions supported by Apache Spark SQL [16].

Algorithm 1: AggFirstJoin

Input: Query Plan Tree QPT
Output: Transformed Query Plan Tree QPT'
$QPT' \leftarrow QPT$
if *QPT has aggregation with join and has not been transformed yet* **then**
  foreach *aggregate expression A in QPT* **do**
    $rule \leftarrow findTransformationRule(A)$
    $A' \leftarrow transform(A, rule)$
    foreach *child $C \in A'$* **do**
      $C' \leftarrow AggFirstJoin(C)$
      replace C with C' in A'
    replace A with A' in QPT'

C. WAN-aware task placement

Since WAN bandwidth is highly constrained and heterogeneous in nature, we try to reduce the network latency by intelligently distributing the join tasks based on the WAN bandwidth available at any DC. We use a mixed integer programming (MIP) optimization to perform the placement of join tasks. The notation used for the optimization is given in Table III.

Item	Notation
Available bandwidth (in Mb/sec) from DC i to DC j	B_{ij}
Fraction of join tasks allocated to DC j	r_j
Size of tables present in DC i	S_i
Minimum data transfer time (in seconds)	z

TABLE III: **Notation for optimization formulation.**

We present the following MIP optimization.

$$\min \quad z \tag{3}$$

$$\text{s.t.,} \quad \sum_j r_j = 1 \tag{4}$$

$$\frac{S_i \cdot r_j}{B_{ij}} \leq z \quad \forall i \neq j \tag{5}$$

The above formulation solves for r_j. It is similar to the formulation used by Iridium [3] but with one major difference: Iridium shuffles the non-aggregated table data across DCs while we shuffle the aggregated data across DCs. Thus, we expect our data shuffle to be significantly faster in comparison to raw data shuffle.

D. Filtering Non-Overlapping Keys

A key would be part of the join result if it exists in all the participating tables. We call such a key as overlapping key. Consequently, those keys which do not form a part of final join result are called as non-overlapping keys[5]. It is unnecessary to shuffle non-overlapping keys across geo-distributed DCs for performing the join operation. To avoid this unnecessary data shuffle, we employ Bloom filters to filter out non-overlapping keys before performing the aggregation and shuffle the aggregated data only for the overlapping keys across geo-distributed DCs.

A Bloom filter is a space-efficient probabilistic data structure that is used to verify whether an element is present in a set or not. It is designed in such a manner that it can give false positives (i.e. an element is present in the set when it is actually not) but it cannot give false negatives (i.e. an element is not present in the set when it is actually present). This serves well for our purpose of filtering non-overlapping keys. For space-efficiency, a Bloom filter is generally implemented as a bit array of N bits. Bloom filters have been used in a wide variety of large scale systems including CDN caching, single DC joins[14].

Since each table is distributed across multiple DCs, we take the following approach to construct a global Bloom filter for filtering out non-overlapping keys:

- At each DC, we first construct a local Bloom filter for each table participating in the join. Each DC sends its local Bloom filters to the central DC.
- At the central DC, we first combine the per partition Bloom filters for each table using OR bitwise operation to get one Bloom filter per table.
- The per table Bloom filters are then combined using AND bitwise operation to construct a global Bloom filter.
- This global Bloom filter is then broadcast to all the DCs.

Although the overhead of Bloom filters in terms of WAN transfer time is negligible because of their space efficiency (i.e. a typical Bloom filter for holding 100K keys with false positive rate of 1% occupies around 100 KB), it can lead to considerable computation overhead due to Bloom filter construction and filtering operations. We evaluate the tradeoff between the WAN usage reduction and computation overhead due to Bloom filters in §**VI**.

Putting everything together:

- Filter out the non-overlapping keys in each table partition at each DC using bloom filters (§**IV-D**).
- Perform aggregation for each filtered table partition using the proposed query transformation approach (§**IV-B**).
- Shuffle the aggregated data among the DCs using a WAN-aware task placement optimization approach (§**IV-C**).

[5]The definition of non-overlapping keys can be modified to accommodate other types of joins such as LEFT and RIGHT joins.

- Send the final join results from each DC to the central DC.

V. Implementation

We implement AggFirstJoin on top of Apache Spark [15], [16], a popular data analytics engine. Our implementation involved four key modules:

Filter Module. This module implements the bloom filter using the open source library [17].

Query Optimization Module. We make changes in Catalyst [18], the default SQL optimizer used by Spark. Catalyst has two phases of query optimization: logical plan optimization and physical plan optimization. We make changes to the logical plan optimization to add rules to identify SELECT-JOIN-GROUPBY blocks in the logical plan and add transformation expressions for pushing the GROUPBY operator before the JOIN operator as per our proposed approach. Our added rules cover common aggregation functions such as SUM, COUNT, AVG, MIN, MAX, VAR etc. paired with any arbitrary transformation expression comprising addition ($+$), subtraction ($-$), multiplication ($\cdot$), and division ($\div$). For queries having functions not covered by AggFirstJoin (See Table II), our system throws a warning to the user about not being able to push aggregation before join. Additionally, our system leaves it up to Catalyst to select the best join order using cost-based optimization in the physical plan optimization.

WAN-aware Task Placement Module. We make changes to the default Spark scheduler and implement the MIP formulation (§IV-C) for WAN-aware task placement. This requires up-to-date bandwidth information across the WAN links connecting the geo-distributed DCs. Each site periodically measures the outgoing bandwidth to every other site, as in [3], and shares it with the scheduler.

VI. Evaluation

A. Experimental Setup

We evaluate AggFirstJoin using a geo-distributed setup comprising eight regions: North California, Ohio, Mumbai, Singapore, Ireland, Frankfurt, Sydney, and Tokyo based on the AWS regions. We keep North California as the central DC where final results are sent. The data is pre-generated and assumed to be available at each site before the query arrives. We split our experiments over two testbeds:

AWS EC2 Testbed. For each AWS region, we instantiate sixteen m4.4xlarge EC2 instances amounting to 256 cores and 1024 GB RAM. We measured the WAN bandwidth between every pair of AWS regions listed above using iperf3 [19]. Our measurement strategy followed prior work [20] wherein we measured the WAN bandwidth periodically every 15 mins for a duration of 24 hours. Experiments in §VI-D are carried out on this testbed.

Emulated Testbed. We also run some of our experiments on a localized CloudLab [21] testbed comprising 16 nodes. Each node had two Intel Xeon Silver 4114 10-core 2.20 GHz CPUs and 192 GB RAM. We assigned two nodes to each region. We emulated the WAN bandwidth heterogeneity (measured on AWS EC2 as explained above) using Linux tc utility [22]. Experiments in §VI-E and are carried out on this testbed.

Evaluation Metrics. We measure and compare the *latency* and *WAN usage* incurred by each evaluated system. Latency is further split into *compute* and *network* (data transfer) latency.

B. Datasets and Queries

We evaluate AggFirstJoin using four datasets:

- **Synthetic Datasets (Syn-U and Syn-Z).** We generate tables having (key, value) tuples with the number of keys (N) in each table ranging from 1K to 160K. For Syn-U dataset, the number of records per key (RPK) follows uniform distribution with RPK ranging from 50 to 2000. For Syn-Z dataset, RPK follows Zipf distribution with Zipf parameter (z) varying from 2.5 to 4 and average RPK ranging from 750 to 1500.
- **TPC-H Benchmark.** The TPC-H benchmark [13] is a popular decision support benchmark consisting of relational tables and queries emulating the activities of any large scale enterprise which manages, sells and distributes products to customers. We use a scale factor of 650 which results in 650 GB data size.
- **AmpLab Big Data Benchmark.** The AmpLab Big Data (ABD) benchmark [23] consists of dataset modelling log files of HTTP server traffic and a collection of random HTML documents mimicking a web crawler. We use a scale factor of 5 which results in 135 GB.

Table IV lists the specific queries for each dataset.

Dataset	Queries
Syn-U	SELECT Key, SUM($C_1 + C_2$) as C_{agg} FROM T_1, T_2 WHERE $T_1.Key == T_2.Key$ GROUP BY Key
Syn-Z	SELECT Key, SUM($C_1 + C_2$) as C_{agg} FROM T_1, T_2 WHERE $T_1.Key == T_2.Key$ GROUP BY Key
TPC-H Query 1	SELECT C.CUSTKEY, C.NAME, SUM(O.TotalPrice) as Revenue FROM Customer C, Orders O WHERE C.CustKey = O.CustKey GROUP BY C.CustKey
TPC-H Query 2	SELECT PS.SuppKey, SUM(LI.Price * (1-LI.Discount) - LI.Quantity * PS.SuppCost) as Profit FROM PartSupplier PS, LineItem LI WHERE PS.SuppKey = LI.SuppKey and PS.PartKey = LI.PartKey GROUP BY PS.SuppKey
AmpLab Big Data	SELECT R.PageUrl, SUM(UV.AdRevenue) as Revenue FROM Rankings R, UserVisits UV WHERE R.PageUrl = UV.DestUrl GROUP BY R.PageUrl

TABLE IV: **Queries used in evaluation.**

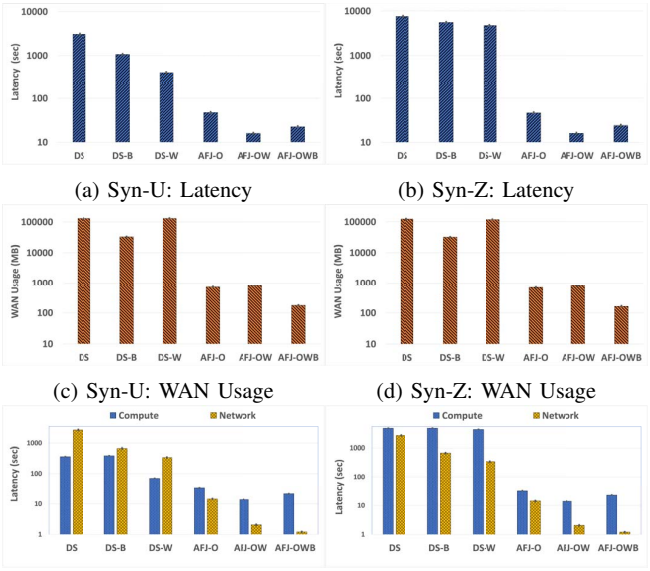
C. Systems for Comparison

We evaluate the following systems:

- **DS**: This approach transfers the joining tables from all the sites to the central DC and then the queries are run on this data using default Spark engine (i.e. Spark without any modifications). This baseline represents centralized aggregation.
- **DS-B**: This approach filters out non-overlapping keys using bloom filters (§IV-D) and sends the filtered data to the central DC where all the join computations are performed using default Spark engine. This baseline represents Approx-Join [14] which also considers bloom filtering to filter out non-overlapping keys for computing joins in an intra-DC environment[6].

[6]We disable ApproxJoin's approximation module as we focus on exact computations.

- **DS-W**: This approach shuffles the raw tables across across geo-distributed sites using the WAN-aware task placement (§IV-C) and then computes join over raw table partitions in a geo-distributed manner. This baseline represents Iridium [3], a widely known GDA technique.
- **AFJ-O**: This approach represents AggFirstJoin but with only query transformation technique (§IV-B) enabled. This approach aggregates each table partition locally using the proposed query transformation and sends the aggregated partitions to the central DC for join computations.
- **AFJ-OW**: This approach represents AggFirstJoin but with only query transformation and WAN-aware task placement enabled. This approach first aggregates the individual table partitions at each site and then shuffles the aggregated partitions across geo-distributed sites based on the WAN-aware task placement.
- **AFJ-OWB**: This approach refers to the entire AggFirstJoin system with all the three techniques (query transformation, WAN-aware task placement and Bloom Filtering) enabled.



(a) Syn-U: Latency (b) Syn-Z: Latency

(c) Syn-U: WAN Usage (d) Syn-Z: WAN Usage

(e) Syn-U: Latency Split (f) Syn-Z: Latency Split

Fig. 3: **Baseline comparison for Syn-U and Syn-Z (Log scale).**

D. Batch Joins: Overall Performance

Syn-U and Syn-Z datasets. Figure 3 compares the latency and WAN usage for different approaches for these two datasets. For both datasets, we fix N=160000, RPK=1500, Overlap = 25 % and for Syn-Z, Z = 4. We study the sensitivity of these parameters in §VI-E.

For latency, AFJ-O gives 9x - 65x and 101x - 162x reduction in latency over the three baselines (DS, DS-B, and DS-W) for Syn-U and Syn-Z respectively. In terms of WAN usage, AFJ-O gives 47x - 187x and 46x - 184x reduction in WAN usage over (DS, DS-B and DS-W) for Syn-U and Syn-Z respectively. AFJ-OW and AFJ-OWB further increase these reductions giving maximum reduction in latency and WAN usage respectively. AFJ-OW gives 26x - 193x and 292x - 468x reduction in latency over (DS, DS-B and DS-W) for Syn-U and Syn-Z respectively. AFJ-OWB gives 188x - 750x and 193x - 757x reduction in WAN usage over (DS, DS-B, and DS-W) for Syn-U and Syn-Z datasets respectively.

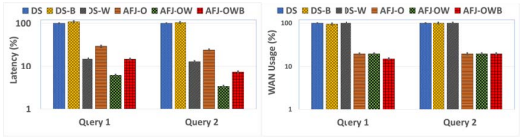
Among AFJ-OW and AFJ-OWB, the results show a tradeoff between latency and WAN usage. AFJ-OWB is able to reduce WAN usage upto 4x more as compared to AFJ-OW by filtering out keys which won't be present in the joined result. But it does so while incurring a computation overhead for Bloom filter construction and filtering operations. Note that the Bloom filter computation and filtering overhead for these two datasets was around 7 seconds which is insignificant in comparison with DS and DS-W.

The massive reduction in latency for AFJ is due to two reasons: reduction in compute latency as well as reduction in network latency. AFJ only shuffles aggregated partitions across WAN as compared to raw table data shuffles in the case of DS, DS-B and DS-W. This is also confirmed by AFJ's corresponding reduction in WAN usage. Although DS-W reduces the latency by WAN-aware task placement, it does not reduce the actual WAN usage as compared to DS because both DS and DS-W are shuffling raw table data across WAN links. DS-B also leads to reduction in network latency and WAN usage because of shuffling only overlapping data. At the same time, AFJ's reduction in latency and WAN usage are orders of magnitude larger as compared to DS-B and DS-W. AFJ's reduction in compute latency is because of join computations over aggregated partitions which limits the size of intermediate Cartesian product as compared to join computations over raw tables which can lead to large sized intermediate Cartesian product table.

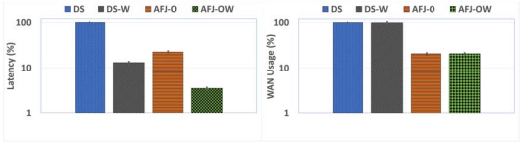
TPC-H and ABD Datasets. Figure 4 plots the latency and WAN usage for these datasets [7]. Similar to Syn-U and Syn-Z, AFJ-OW and AFJ-OWB give maximum reduction in latency and WAN usage respectively for TPC-H. AFJ-OW gives 2.5x - 31x reduction in latency while AFJ-OWB gives 5x - 6x reduction in WAN Usage over (DS, DS-B and DS-W). For ABD, AFJ-OW gives 3.8x - 28.5x reduction in latency and 4.8x reduction in WAN usage over (DS and DS-W). AFJ's reduction for these two datasets are significant but not as much as Syn-Z and Syn-U because of lower RPK and one-to-many joins. The RPK for these datasets range from 1 to 40 which limits the aggregation opportunity for AFJ. Additionally, the joins involved are all one-to-many in nature and hence, even with raw table data, the size of intermediate joined table is not larger than the size of original tables combined and hence, there is limited opportunity for AFJ to reduce the compute latency.

Multi-Way Joins. Figure 5 compares the latency and WAN usage for multi-way joins. As the number of joining tables increase from 2 to 5, we make the following observations. The total latency increases exponentially (up to 2000x) for DS,

[7]For ABD, we do not show the DS-B and AFJ-OWB baselines since the generated joining tables always have 100% overlap w.r.t. the join key. Hence, it is known beforehand that filtering would never help.



(a) TPC-H: Latency (b) TPC-H: WAN Usage



(c) ABD: Latency (d) ABD: WAN Usage

Fig. 4: **Baseline comparison for TPC-H and ABD (Log scale)**

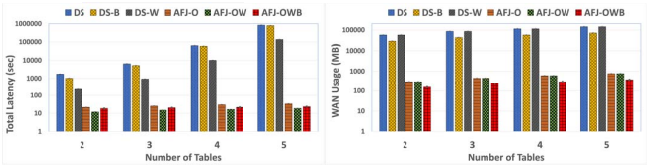


Fig. 5: **Multi-way joins (Log scale)**

DS-B and DS-W while it only marginally increases for AFJ-O, AFJ-OW and AFJ-OWB (up to 1.6x). This is expected since the size of the intermediate joined table (Cartesian product) increases exponentially with the increase in the number of tables for many-to-many joins, thus increasing the compute latency proportionally. On the other hand, AFJ first aggregates each table and hence, the size of intermediate joined table does not increase even with the increase in the number of joining tables. We find that the increase in network latency is linear for all the approaches. This is also expected since the amount of data to be shuffled across WAN is directly proportional to the number of tables shuffled and their corresponding sizes. At the same time, AFJ's increase in network latency is insignificant as compared to other baselines. For the same reason, the WAN usage for AFJ is also significantly lower (upto 2000x lower) as compared to DS, DS-B and DS-W for multi-way joins.

E. Batch Joins: Sensitivity Analysis

Variation of Overlap %. Figure 6 compares the latency and WAN usage for varying overlap % (% of total number of keys which will be present in the joined result) for Syn-U dataset (RPK = 1500 and N = 160K). As the overlap % increases from 5 % to 100 %, we note that the latency for DS, DS-B and DS-W increases linearly with the increase in overlap %. The increase in latency is more pronounced for DS-B since at lower overlap %, DS-B is able to reduce the network latency by filtering out the non-overlapping keys. On the other hand, DS and DS-W don't do any such filtering. This is also evident from the WAN usage for these baselines. The WAN usage remains almost same for DS and DS-W while increases linearly for DS-B.

We don't observe any significant changes in latency and WAN usage for AFJ-O and AFJ-OW since these approaches don't perform any filtering. AFJ-OWB, on the other hand, sees a linear increase in the WAN usage as expected. The latency also increases for AFJ-OWB with increase in overlap % but only marginally. We conclude that at lower overlap %, AFJ-OWB can give significantly more reduction in WAN usage as compared to AFJ-OW, at the cost of marginal increase in latency. As the overlap % increases, the additional reduction in WAN usage provided by AFJ-OWB goes on decreasing until it does not provide any advantage for the latency overhead (due to bloom filters) over AFJ-OW.

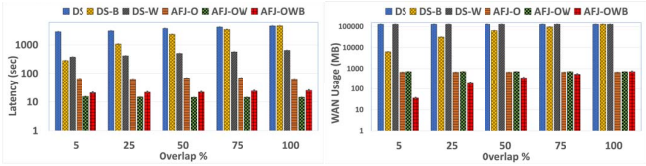


Fig. 6: **Variation of Overlap % (Log scale)**

Variation of RPK. Figure 7 compares the latency and WAN usage for varying records-per-key (RPK) values for Syn-U dataset (Overlap % $= 50\%$ and N=40K). As the average RPK increases from 1 to 1500, we note that the latency and WAN usage increase by 90x - 175x and 1500x respectively for DS, DS-B and DS-W. On the contrary, AFJ-O, AFJ-OW and AFJ-OWB see only an insignificant increase in latency (only 3.5x for AFJ-OW) and no increase in WAN usage due to performing aggregation before data shuffle and join computations.

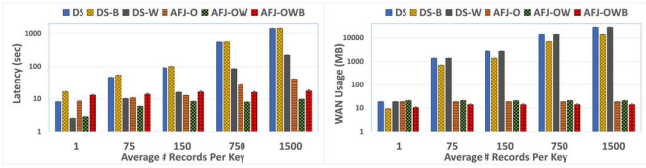


Fig. 7: **Variation of RPK (Log scale)**

Variation of Number of Keys. Figure 8 compares the latency and WAN usage for varying number of keys in the Syn-U dataset. As the number of keys in the joining tables increase from 4K to 40K, we note that AFJ-OW and AFJ-OWB are always significantly better than DS, DS-B and DS-W in terms of both latency and WAN usage. Moreover, the reductions in latency provided by AFJ-OW and AFJ-OWB in comparison to DS, DS-B and DS-W increase with increasing number of keys. For instance, AFJ-OW's reduction in latency over DS-W increase from 6x to 22x with increasing N. The increase in WAN usage is almost linear for all the approaches but remains very low for AFJ-O, AFJ-OW and AFJ-OWB.

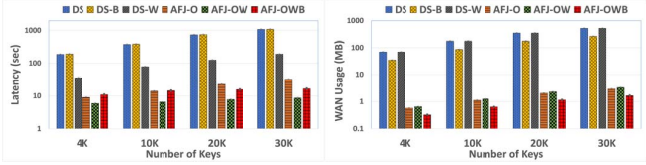


Fig. 8: **Variation of number of keys (Log scale)**

Variation of Skew. Figure 8 compares the latency and WAN usage for varying skew (Zipf parameter) in the Syn-Z dataset.

As the value of Zipf parameter changes from 4 to 2.5 (i.e. the skew is increased), we note that the latency for (DS, DS-B, and DS-W) increases by a factor of 20 while the WAN usage does not show any change. This is because these approaches spend more time in join computations for keys with high RPK (i.e. more frequent keys) as the skew in RPK distribution increases. On the other hand, the latency and WAN usage for (AFJ-O, AFJ-OW, and AFJ-OWB) do not show any noticeable increase since these approaches first aggregate and then compute the join on aggregated values.

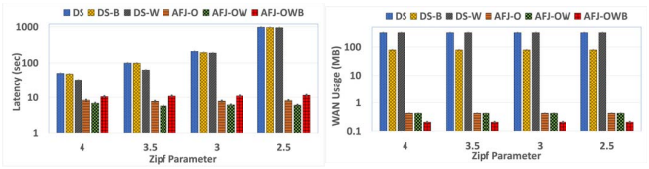


Fig. 9: **Variation of skew in RPK distribution (Lower Zipf signifies higher skew) (Log scale)**

VII. Related Work

Systems for Big Data Analytics. A number of data processing engines [15], [24], [25], [26], [27], [28], [29] have been proposed for efficiently processing big data workloads and enabling useful analytics. But all of these systems are primarily designed and engineered for intra-DC environments where the network bandwidth as well as compute resources are plenty. Hence, these systems cannot be used directly in WAN environment as also confirmed by prior work [3], [12], [1], [5]. We build our prototype on top of Apache Spark, a very popular analytics engine.

Geo-Distributed Analytics. To address the issue of constrained and heterogeneous compute and network bandwidth resources in WAN environments, geo-distributed analytics has been proposed which processes the data in a geographically distributed manner instead of transferring all the data to a centralized DC and processing it in a centralized manner. Systems such as Iridium [3], Tetrium [2], Clarinet [1], WANanlytics [5], Bohr [30], Kimchi [4], Yugong [9] and Heintz et al [31] handle batch workloads while systems such as JetStream [32], AWStream [33], Sana [12], WASP [34], Heintz et al [7], [35], AggNet [36] and Kumar et al [6] support streaming workloads in WAN environments. These systems propose WAN-aware data and task placement as well as job scheduling for minimizing metrics such as query latency, WAN usage, accuracy and cost. Clarinet [1] additionally also proposes WAN-aware query optimization and multi-query scheduling. In the context of geo-distributed joins, these systems shuffle raw data across geo-distributed sites and perform the join on the raw tables or streams without any prior aggregation. This can lead to high compute and network latency as well as high WAN usage as shown in this work. Our proposed system, AggFirstJoin, reduces all the three aforementioned metrics by pre-aggregating the tables before shuffle and join computations. Moreover, our work is complementary to the optimizations proposed by these systems. For instance, we augmented our query transformation approach with WAN-aware task placement similar to Iridium.

Database Joins and Query Optimization. Database join is one of the most fundamental building blocks in analytical queries and hence, it has been studied in great detail in the past [37], [38], [39], [40]. Most of these optimizations are suited only for intra-DC environments for single node or multi-node databases. They are not sufficient for efficient join computations in WAN environment as shown by the existing GDA systems (discussed above). Pushing group by before join has been studied in the past in the context of single-server databases [10], [11] where there is only one partition per table. In contrast, we propose query transformation for geo-distributed environment where we provide solution to performing join on geo-distributed partitions of the joining tables after performing geo-distributed aggregation. Additionally, we integrate WAN constraints, heterogeneity and WAN-aware filtering with this aggregation pushdown. Moreover, the existing techniques only consider aggregation operations over single table columns and supports only real number data types while our technique works for aggregation over derived columns which may be derived from columns spanning multiple tables and supports more generic data types such as strings. Finally, the existing techniques optimize only a limited set of aggregation functions such as Sum, Min, Max, Average while our technique optimizes a broader set of aggregation functions including higher order functions such as Variance, Skewness, Kurtosis etc. (Table II). ApproxJoin [14] proposed approximate distributed joins using bloom filters and stratified sampling. While ApproxJoin still computes aggregation after the join operation, we propose query transformation which allows us to perform aggregation before join and thus, reduce the compute and network latency associated with joins. We show in §VI that our approach performs better than ApproxJoin (without sampling) in terms of latency and WAN usage.

Bloom Filters. Bloom filters [41] have been used in applications such as web caching in CDNs [42], peer-to-peer overlay networks [43], database joins [44], [45], [14]. While our bloom filter usage in AggFirstJoin is similar to existing work, our work differs in two ways: (1) we apply bloom filters in a geo-distributed environment where they have not been used before (2) we combine bloom filters with query transformation and WAN-aware task placement to build a holistic system for geo-distributed joins.

VIII. Conclusion

We proposed *AggFirstJoin* to minimize the cost of geo-distributed joins using a query transformation technique which pushes (a transformed) aggregation before join in a manner to produce the same results as the original query. We further augmenteded with a WAN-aware task placement and a Bloom filtering approach. Our evaluations showed our proposed technique achieves upto 300x reduction in query execution time and 200x reduction in WAN usage as compared to state-of-the-art GDA techniques.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for many constructive comments and suggestions. This work was sponsored in part by NSF under Grants CNS-1717834 and CNS-1717179, as well as by DARPA contract HR001117C0049.

REFERENCES

[1] R. Viswanathan, G. Ananthanarayanan, and A. Akella, "Clarinet: Wan-aware optimization for analytics queries," ser. OSDI'16. USA: USENIX Association, 2016, p. 435–450.

[2] C.-C. Hung, G. Ananthanarayanan, L. Golubchik, M. Yu, and M. Zhang, "Wide-area analytics with multiple resources," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190508.3190528

[3] Q. Pu, G. Ananthanarayanan, P. Bodik, S. Kandula, A. Akella, P. Bahl, and I. Stoica, "Low latency geo-distributed data analytics," ser. SIGCOMM '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 421–434. [Online]. Available: https://doi.org/10.1145/2785956.2787505

[4] K. Oh, A. Chandra, and J. B. Weissman, "A network cost-aware geo-distributed data analytics system," in *20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID 2020, Melbourne, Australia, May 11-14, 2020*. IEEE, 2020, pp. 649–658. [Online]. Available: https://doi.org/10.1109/CCGrid49817.2020.00-28

[5] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, K. Karanasos, J. Padhye, and G. Varghese, "Wanalytics: Geo-distributed analytics for a data intensive world," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1087–1092. [Online]. Available: https://doi.org/10.1145/2723372.2735365

[6] D. Kumar, J. Li, A. Chandra, and R. Sitaraman, "A ttl-based approach for data aggregation in geo-distributed streaming analytics," in *ACM SIGMETRICS*, New York, NY, USA, Jun. 2019. [Online]. Available: https://doi.org/10.1145/3341617.3326144

[7] B. Heintz, A. Chandra, and R. K. Sitaraman, "Optimizing grouped aggregation in geo-distributed streaming analytics," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 133–144. [Online]. Available: https://doi.org/10.1145/2749246.2749276

[8] R. Hong and A. Chandra, "Dlion: Decentralized distributed deep learning in micro-clouds," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 227–238. [Online]. Available: https://doi.org/10.1145/3431379.3460643

[9] Y. Huang, Y. Shi, Z. Zhong, Y. Feng, J. Cheng, J. Li, H. Fan, C. Li, T. Guan, and J. Zhou, "Yugong: Geo-distributed data and job placement at scale," *Proc. VLDB Endow.*, vol. 12, no. 12, p. 2155–2169, Aug. 2019. [Online]. Available: https://doi.org/10.14778/3352063.3352132

[10] W. P. Yan and P. Larson, "Eager aggregation and lazy aggregation," in *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, U. Dayal, P. M. D. Gray, and S. Nishio, Eds. Morgan Kaufmann, 1995, pp. 345–357. [Online]. Available: http://www.vldb.org/conf/1995/P345.PDF

[11] S. Chaudhuri and K. Shim, "Including group-by in query optimization," in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, p. 354–366.

[12] A. Jonathan, A. Chandra, and J. Weissman, "Multi-query optimization in wide-area streaming analytics," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 412–425. [Online]. Available: https://doi.org/10.1145/3267809.3267842

[13] TPC-H Benchmark, Accessed: 2021-05-26, http://www.tpc.org/tpch/.

[14] D. L. Quoc, I. E. Akkus, P. Bhatotia, S. Blanas, R. Chen, C. Fetzer, and T. Strufe, "Approxjoin: Approximate distributed joins," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 426–438. [Online]. Available: https://doi.org/10.1145/3267809.3267834

[15] 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*, ser. NSDI'12. USA: USENIX Association, 2012, p. 2.

[16] 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 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[17] Bloom Filters, Accessed: 2021-05-17, https://github.com/alexandrnikitin/bloom-filter-scala.

[18] 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 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1383–1394. [Online]. Available: https://doi.org/10.1145/2723372.2742797

[19] iperf3, Accessed: 2021-05-17, https://software.es.net/iperf/.

[20] F. Lai, M. Chowdhury, and H. Madhyastha, "To relay or not to relay for inter-cloud transfers?" in *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*. Boston, MA: USENIX Association, Jul. 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/lai

[21] CloudLab, Accessed: 2021-05-17, https://www.cloudlab.us.

[22] Linux traffic control, Accessed: 2020-08-30, https://man7.org/linux/man-pages/man8/tc.8.html.

[23] AmpLab Big Data Benchmark, Accessed: 2021-05-26, https://amplab.cs.berkeley.edu/benchmark/.

[24] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at linkedin," *Proc. VLDB Endow.*, vol. 10, no. 12, p. 1634–1645, Aug. 2017. [Online]. Available: https://doi.org/10.14778/3137765.3137770

[25] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "Millwheel: Fault-tolerant stream processing at internet scale," *Proc. VLDB Endow.*, vol. 6, no. 11, p. 1033–1044, Aug. 2013. [Online]. Available: https://doi.org/10.14778/2536222.2536229

[26] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 423–438. [Online]. Available: https://doi.org/10.1145/2517349.2522737

[27] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache flink^{TM}: Stream and batch processing in a single engine," *IEEE Data Eng. Bull.*, vol. 38, no. 4, pp. 28–38, 2015. [Online]. Available: http://sites.computer.org/debull/A15dec/p28.pdf

[28] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 239–250. [Online]. Available: https://doi.org/10.1145/2723372.2742788

[29] KSQL: Streaming SQL for Kafka., Accessed: 2018-10-29, https://www.confluent.io/product/ksql/.

[30] H. Li, H. Xu, and S. Nutanong, "Bohr: Similarity aware geo-distributed data analytics," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 267–279. [Online]. Available: https://doi.org/10.1145/3281411.3281418

[31] B. Heintz, A. Chandra, R. K. Sitaraman, and J. Weissman, "End-to-end optimization for geo-distributed mapreduce," *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 293–306, 2016.

[32] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in jetstream: Streaming analytics in the wide area," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association,

Apr. 2014, pp. 275–288. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin

[33] B. Zhang, X. Jin, S. Ratnasamy, J. Wawrzynek, and E. A. Lee, "Awstream: Adaptive wide-area streaming analytics," ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 236–252. [Online]. Available: https://doi.org/10.1145/3230543.3230554

[34] A. Jonathan, A. Chandra, and J. Weissman, "Wasp: Wide-area adaptive stream processing," in *Proceedings of the 21st International Middleware Conference*, ser. Middleware '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 221–235. [Online]. Available: https://doi.org/10.1145/3423211.3425668

[35] B. Heintz, A. Chandra, and R. K. Sitaraman, "Trading timeliness and accuracy in geo-distributed streaming analytics," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 361–373. [Online]. Available: https://doi.org/10.1145/2987550.2987580

[36] D. Kumar, S. Ahmad, A. Chandra, and R. K. Sitaraman, "Aggnet: Cost-aware aggregation networks for geo-distributed streaming analytics," in *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 2021.

[37] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endow.*, vol. 9, no. 3, p. 204–215, Nov. 2015. [Online]. Available: https://doi.org/10.14778/2850583.2850594

[38] Y. E. Ioannidis, "Query optimization," *ACM Computing Surveys (CSUR)*, vol. 28, no. 1, pp. 121–123, 1996.

[39] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization." in *VLDB*, vol. 97, 1990, pp. 314–325.

[40] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, "Query optimization for massively parallel data processing," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, 2011, pp. 1–13.

[41] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[42] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, p. 52–66, Jul. 2015. [Online]. Available: https://doi.org/10.1145/2805789.2805800

[43] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[44] L. F. Mackert and G. M. Lohman, "R* optimizer validation and performance evaluation for local queries," in *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '86. New York, NY, USA: Association for Computing Machinery, 1986, p. 84–95. [Online]. Available: https://doi.org/10.1145/16894.16863

[45] B. Ding, S. Chaudhuri, and V. Narasayya, "Bitvector-aware query optimization for decision support queries," ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 2011–2026. [Online]. Available: https://doi.org/10.1145/3318464.3389769