Chasing Similarity: Distribution-aware Aggregation Scheduling

Feilong Liu[1], Ario Salmasi[1], Spyros Blanas[1], Anastasios Sidiropoulos[2]
[1]The Ohio State University, [2]University of Illinois at Chicago
{liu.3222,salmasi.1,blanas.2}@osu.edu, sidiropo@gmail.com

ABSTRACT

Parallel aggregation is a ubiquitous operation in data analytics that is expressed as GROUP BY in SQL, reduce in Hadoop, or segment in TensorFlow. Parallel aggregation starts with an optional local pre-aggregation step and then repartitions the intermediate result across the network. While local pre-aggregation works well for low-cardinality aggregations, the network communication cost remains significant for high-cardinality aggregations even after local pre-aggregation. The problem is that the repartition-based algorithm for high-cardinality aggregation does not fully utilize the network.

In this work, we first formulate a mathematical model that captures the performance of parallel aggregation. We prove that finding optimal aggregation plans from a known data distribution is NP-hard, assuming the Small Set Expansion conjecture. We propose GRASP, a GReedy Aggregation Scheduling Protocol that decomposes parallel aggregation into phases. GRASP is distribution-aware as it aggregates the most similar partitions in each phase to reduce the transmitted data size in subsequent phases. In addition, GRASP takes the available network bandwidth into account when scheduling aggregations in each phase to maximize network utilization. The experimental evaluation on real data shows that GRASP outperforms repartition-based aggregation by 3.5× and LOOM by 2.0×.

PVLDB Reference Format:
Feilong Liu, Ario Salmasi, Spyros Blanas, Anastasios Sidiropoulos. Chasing Similarity: Distribution-aware Aggregation Scheduling. *PVLDB*, 12(3): 292-306, 2018.
DOI: https://doi.org/10.14778/3291264.3291273

1. INTRODUCTION

Aggregation is widely used in data analytics. Parallel aggregation is executed in two steps. The first step is an optional local aggregation where data is aggregated locally, followed by a second step where data is repartitioned and transferred to the final destination node for aggregation [45, 14]. The local aggregation can reduce the amount of data transferred in the second step for algebraic aggregations, as tuples with the same GROUP BY key are aggregated to a single tuple during local aggregation [6, 52, 22, 35, 48]. Local aggregation works effectively for low-cardinality domains, such as age, sex or country, where data can be reduced substantially and make the cost of the repartition step negligible. However, high-cardinality aggregations see little or no benefit from local aggregation. Optimizing the repartitioning step for high-cardinality aggregations has received less research attention.

High-cardinality aggregations are surprisingly common in practice. One example is sessionization, where events in a timestamp-ordered log need to be grouped into user sessions for analysis. An exemplar is the publicly-available Yelp dataset where 5.2M reviews are aggregated into 1.3M user sessions [53]. Even when there are no high-cardinality attributes, aggregation on composite keys of multiple attributes can lead to high-cardinality aggregations, which is common in data cube calculations [16].

This paper focuses on reducing the communication cost for high-cardinality aggregations. We classify aggregations into two types: all-to-one aggregation and all-to-all aggregation. In all-to-one aggregation, one coordinator collects and aggregates data from all compute nodes. All-to-one aggregation frequently happens at the last stage of a query. In all-to-all aggregation, data is repartitioned on the GROUP BY attributes and every node aggregates a portion of the data. All-to-all aggregation is common in the intermediate stages of a query plan.

Directly transmitting the data to the destination node during an aggregation underutilizes the network. In all-to-one aggregation, the receiving link of the destination is the bottleneck while every other receiving link in the network is idle. In all-to-all aggregation, workload imbalance due to skew or non-uniform networks [17, 27] means that some network links will be underutilized when waiting for the slower or overburdened links to complete the repartitioning.

Systems such as Dremel [32], Camdoop [7], NetAgg [31] and SDIMS [51] reduce the communication cost and increase network utilization by using *aggregation trees* for all-to-one aggregations. The most relevant prior work is LOOM [8, 9], which builds aggregation trees in a network-aware manner. LOOM assumes that every node stores $|R_{leaf}|$ distinct keys and that the cardinality of the final aggregation result is $|R_{root}|$. Given these parameters as input, LOOM produces an aggregation tree with a fan-in that is a function of the reduction rate $|R_{root}|/|R_{leaf}|$. Applying LOOM during query execution is not trivial, however, as the cardinality of the input and the final result is not known in advance. (Even

Proceedings of the VLDB Endowment, Vol. 12, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3291264.3291273

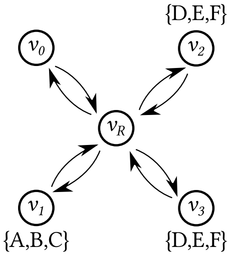


Figure 1: Graph representation of a cluster with four nodes. The aggregation destination is v_0 and the router is v_R.

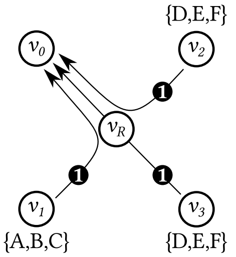


Figure 2: Aggregation based on **repartitioning** completes in 9 time units. The bottleneck is the $v_R \rightarrow v_0$ link.

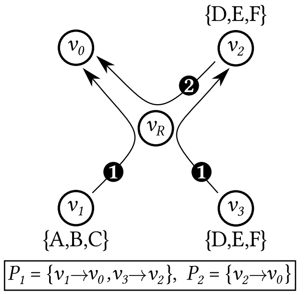

{D,E,F}
v_0
v_2
2
v_R
1
1
v_1
v_3
{A,B,C}
{D,E,F}
$P_1 = \{v_1 \rightarrow v_0, v_3 \rightarrow v_2\},\ P_2 = \{v_2 \rightarrow v_0\}$

Figure 3: The **similarity-aware** plan completes in 6 time units.

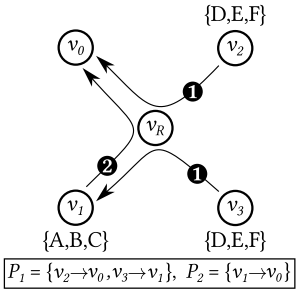

{D,E,F}
v_0
v_2
1
v_R
2
1
v_1
v_3
{A,B,C}
{D,E,F}
$P_1 = \{v_2 \rightarrow v_0, v_3 \rightarrow v_1\},\ P_2 = \{v_1 \rightarrow v_0\}$

Figure 4: The **similarity-oblivious** plan finishes in 9 time units.

estimations of the cardinality can be inaccurate [24].) Furthermore, the aggregation plan that LOOM produces fails to consider how the *similarity between partitions* impacts the reduction rate at intermediate steps of the aggregation.

The importance of considering partition similarity during aggregation can be shown with an example. Figure 1 shows an all-to-one aggregation in a 4-node cluster, where v_R is the switch, node v_0 is the destination node, node v_1 stores three tuples with keys A, B and C, and nodes v_2 and v_3 store three tuples each with keys D, E and F. (For simplicity, the figures only show the GROUP BY keys.)

- The **repartitioning** strategy in Figure 2 finishes the aggregation in 9 time units, where one time unit is the time v_0 needs to receive and process a single tuple.
- The **similarity-aware** aggregation plan in Figure 3 proceeds in two phases. In the first phase, v_1 transmits keys {A,B,C} to v_0 and v_3 transmits keys {D,E,F} to v_2. In the second phase, v_2 computes the partial aggregation and transmits keys {D,E,F}. The entire aggregation completes in 6 time units — 1.5× faster than repartitioning.
- The **similarity-oblivious** aggregation plan shown in Figure 4 transmits keys {D,E,F} from v_3 to v_1 in the first phase and then needs 6 time units in the second phase to transmit keys {A,B,C,D,E,F} to v_0. The entire aggregation completes in 9 time units, as fast as repartitioning.

This paper introduces GRASP, an algorithm that carefully constructs aggregation plans to accelerate high-cardinality aggregation. Unlike prior solutions [32, 7, 31, 51] that do not consider if data can be combined during an aggregation, GRASP *aggregates fragments with similar keys first to improve performance.* GRASP has the following attributes: (1) it is distribution-aware as it selects aggregation pairs that will produce smaller partial aggregates, (2) it is topology-aware as it schedules larger data transfers on faster network links, (3) it achieves high network utilization as it uses as many network links as possible.

The paper is structured as follows. Section 2 develops a theoretical model for the network cost of parallel data aggregation. Section 3 introduces GRASP, a topology-aware and data distribution-aware algorithm, that accelerates aggregations by leveraging partition similarity. A natural question to ask is if GRASP produces aggregation plans that approximate the optimal plan by some constant factor. Section 4 proves that the aggregation scheduling problem cannot be approximated within a constant factor by any polynomial algorithm (including GRASP), assuming the SSE conjecture. Section 5 contains the experimental evaluation which shows that GRASP can be up to 3.5× faster than repartitioning and up to 2.0× faster than LOOM on real datasets.

2. PROBLEM DEFINITION

We use a connected, directed, weighted graph $G = (V(G), E(G))$ to represent the network topology of the cluster. Each edge $\langle v_i, v_j \rangle \in E(G)$ represents one network link, with the edge direction to be the direction of data flow.

The fat-tree topology is widely used in data centers [1]. We represent all routers in the network as a single node $v_R \in V(G)$ and model the fat-tree topology as a star network. The set $V_C = V(G) - \{v_R\}$ represents the compute nodes of the cluster. Compute nodes have bidirectional network links, therefore $E(G) = \{\langle s, v_R \rangle | s \in V_C\} \bigcup \{\langle v_R, t \rangle | t \in V_C\}$, where edge $\langle s, v_R \rangle$ represents the uplink and edge $\langle v_R, t \rangle$ represents the downlink.

2.1 Modeling all-to-one aggregations

Aggregation Model. We first consider an aggregation where data is aggregated to one single node $v^* \in V_C$. The aggregation consists of multiple phases which execute in serial order. We use $\mathbb{P}$ to denote an aggregation execution plan with n phases, $\mathbb{P} = \{P_1, P_2, ..., P_n\}$, where P_i represents one phase of the aggregation. In a phase P_i, there are k concurrent data transfers, $P_i = \{s_1 \rightarrow t_1, ..., s_k \rightarrow t_k\}$, where $s_j \rightarrow t_j$ denotes the data transfer in which node s_j sends all its data to node t_j. Figure 3 shows an aggregation execution plan $\mathbb{P}$ with two phases P_1 and P_2. Phase P_1 performs two data transfers $v_1 \rightarrow v_0, v_3 \rightarrow v_2$, and phase P_2 performs one data transfer $v_2 \rightarrow v_0$.

We impose one constraint in the selection of $s \rightarrow t$ pairs: node s will never send its data to a node t that has no data, unless t is the final destination node v^*, as no data will be aggregated in this case. (In fact, we could not find any instance where transferring to an empty node t would be beneficial over transmitting data directly to the destination v^* in a single-path star topology.) Hence, a node can be a receiver multiple times across multiple phases, but once it transmits its data in some phase P_i it becomes inactive and it will

Table 1: Symbol definitions.

Symbol	Description
$s \rightarrow t$	Data transfer from node s to node t
P_i	Phase i, $P_i = \{s_1 \rightarrow t_1,\ s_2 \rightarrow t_2,\ \ldots\}$
$\mathbb{P}$	Aggregation plan, $\mathbb{P} = \{P_1, P_2, \ldots\}$
$X_i^l(v)$	Data of partition l in node v after P_i completes
$X_i(v)$	Data in v after P_i finishes, $X_i(v) = \bigcup_l X_i^l(v)$
$X_0(v)$	Data in v before the aggregation starts
$Y_i(s \rightarrow t)$	Data sent from s to t in phase P_i
w	Size of one tuple
$B(s \rightarrow t)$	Available bandwidth for the $s \rightarrow t$ data transfer
$\texttt{COST}(s \rightarrow t)$	Network cost for the $s \rightarrow t$ data transfer

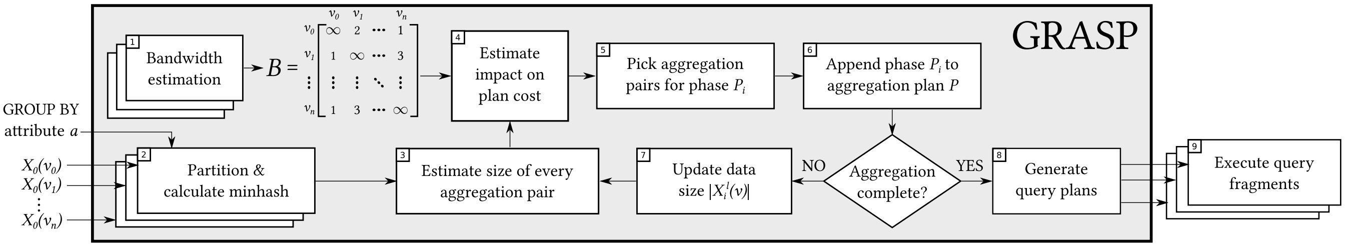

GRASP
1 Bandwidth estimation
B =
v0 v1 vn
∞ 2 ⋯ 1
1 ∞ ⋯ 3
1 3 ⋯ ∞
4 Estimate impact on plan cost
5 Pick aggregation pairs for phase Pi
6 Append phase Pi to aggregation plan P
GROUP BY attribute a
X0(v0)
X0(v1)
X0(vn)
2 Partition & calculate minhash
3 Estimate size of every aggregation pair
7 Update data size |Xil(v)|
NO
Aggregation complete?
YES
8 Generate query plans
9 Execute query fragments

Figure 5: The GRASP framework.

not participate in the aggregation in phases $P_{i+1}, ..., P_n$. A corollary is that a node cannot be both sending and receiving data in the same phase.

Let $X_0\left(v\right)$ be the data in node v in the beginning of the aggregation execution and $X_i\left(v\right)$ be the data in node v after phase P_i completes. Let $Y_i(s \rightarrow t)$ be the data sent from s to t in phase P_i. A node will send all its local data within one phase, hence $Y_i(s \rightarrow t) = X_{i-1}(s)$. After phase P_i completes, for every transfer $s \rightarrow t \in P_i$, $X_i\left(s\right) = \varnothing$ and

$$X_i\left(t\right) = X_{i-1}\left(t\right) \bigcup \left(\bigcup_{s \rightarrow t \in P_i} X_{i-1}\left(s\right) \right) \tag{1}$$

The aggregation has finished in phase n when all nodes except v^* have sent their data out for aggregation:

$$\forall v \in \left\{V_C - \left\{v^*\right\}\right\} : X_n\left(v\right) = \varnothing \tag{2}$$

Aggregation cost. The aggregation execution plan $\mathbb{P} = \{P_1, ..., P_n\}$ consists of phases in serial order. Hence the network cost of $\mathbb{P}$ is:

$$\texttt{COST}\left(\mathbb{P}\right) = \sum \texttt{COST}\left(P_i\right) \tag{3}$$

The network cost for phase $P_i = \{s_1 \rightarrow t_1, ..., s_k \rightarrow t_k\}$ is the cost of the network transfer which completes last:

$$\texttt{COST}\left(P_i\right) = \underset{s_j \rightarrow t_j \in P_i}{max} \texttt{COST}\left(s_j \rightarrow t_j\right) \tag{4}$$

The cost of the data transfer $s_j \rightarrow t_j$ is the time it takes to transfer $|Y_i(s_j \rightarrow t_j)|$ tuples of size w each over the available link bandwidth $B\left(s_j \rightarrow t_j\right)$:

$$\texttt{COST}\left(s_j \rightarrow t_j\right) = \frac{|Y_i(s_j \rightarrow t_j)| \cdot w}{B\left(s_j \rightarrow t_j\right)} \tag{5}$$

Section 3.2 shows how GRASP estimates $B\left(s_j \rightarrow t_j\right)$ without network topology information. Section 4.1 shows one way to calculate $B\left(s_j \rightarrow t_j\right)$ if all network activity is known.

Problem definition. Given a connected, directed, weighted graph G, the data $X_0\left(v\right)$ in every node $v \in V_C$, the final destination node $v^* \in V_C$, obtain an aggregation execution plan containing one or more phases $\mathbb{P} = \{P_1, P_2, ..., P_n\}$ such that $\texttt{COST}\left(\mathbb{P}\right)$ is minimized.

2.2 Modeling all-to-all aggregations

The all-to-all aggregation model executes multiple all-to-one aggregations over different partitions in a single plan.

Aggregation Model. In all-to-all aggregation data is divided into m partitions, $L = \{l_1, l_2, ..., l_m\}$. Every compute node in V_C is the aggregation destination for one or more partitions. This is specified by a mapping $M : L \rightarrow V_C$ that maps a partition $l \in L$ to a specific destination $v \in V_C$. Let $X_0^l\left(v\right)$ be the data of partition l in node v in the beginning of the aggregation execution and $X_i^l\left(v\right)$ be the data of partition l in node v after phase P_i completes.

Within one aggregation phase, a node s will send an entire partition l of local data to t, hence $Y_i(s \rightarrow t) = X_{i-1}^l(s) \subseteq X_{i-1}(s)$. Once a node transmits all its data for partition l it becomes inactive in subsequent phases for this partition, but it will participate in aggregations for other active partitions. Hence, in all-to-all aggregation a node can be both sending and receiving data in the same phase, as long as it does not send and receive data belonging to the same partition. $X_i\left(v\right)$ is the data in node v after phase P_i completes:

$$X_i\left(v\right) = X_{i-1}\left(v\right) \bigcup \left(\bigcup_{s \rightarrow v \in P_i} Y_i\left(s \rightarrow v\right) \right) - \bigcup_{v \rightarrow t \in P_i} Y_i(v \rightarrow t) \tag{6}$$

All-to-all aggregation completes when data in all partitions are aggregated to their corresponding destination:

$$\forall\ l \rightarrow v^* \in M : \forall v \in \left\{V_C - \left\{v^*\right\}\right\} : X_n^l\left(v\right) = \varnothing \tag{7}$$

Problem definition. Given a connected, directed, weighted graph G, the data $X_0^l\left(v\right)$ for each partition $l \in L$ in every node $v \in V_C$, and a mapping $M : L \rightarrow V_C$ denoting the destination of each partition, obtain an aggregation execution plan containing one or more phases $\mathbb{P} = \{P_1, P_2, ..., P_n\}$ such that $\texttt{COST}\left(\mathbb{P}\right)$ is minimized.

3. THE GRASP FRAMEWORK

This section introduces GRASP, a greedy aggregation scheduling protocol, which uses partition similarity as a heuristic to carefully schedule data transfers to improve performance.

3.1 Overview

Figure 5 shows an overview of the GRASP framework. The inputs to the framework are the data $X_0(v)$ in every node v and the Group By attribute a. The input data may be either a table in the database or an intermediate result produced during query processing. Steps 1, 2 and 9 are run by all compute nodes, while steps 3–8 are run in the coordinator.

1) **Bandwidth estimation.** Every node estimates the available bandwidth between itself and other nodes and stores it in matrix B. Section 3.2 describes the process in detail.

2) **Partition, pre-aggregate and calculate minhash signatures.** Every node partitions and aggregates data locally. During this operation, every node runs the minhash algorithm [3, 21, 13] to produce succinct minhash signatures.

3) **Estimate the cardinality of every possible pair.** The coordinator collects the minhash signatures and estimates the cardinality of all possible aggregation pairs. An

aggregation pair is a partition l, a source node s and a destination node t. Section 3.3 presents the algorithms in detail.

4) **Estimate the cost of the final plan.** The coordinator uses the available bandwidth matrix B as input and estimates the runtime cost and the future benefit of executing every possible aggregation pair. Section 3.4 describes the cost heuristic.

5) **Generate aggregation phase P_i.** The coordinator selects aggregation pairs for phase P_i based on their cost. The detailed algorithm is described in Section 3.5.

6) **Add P_i to aggregation plan $\mathbb{P}$.** If the aggregation is complete, the aggregation plan $\mathbb{P}$ is scheduled for execution.

7) **Update data size $|X_i^l(v)|$.** The coordinator updates the estimation of the size of each partition $|X_i^l(v)|$ in every node for the next phase of the aggregation. GRASP does not make another pass over the data, as the minhash signature of any intermediate result can be calculated from the original minhash signatures obtained in Step 2.

8) **Generate query plans.** The aggregation planning is complete. GRASP generates query plans for execution.

9) **Query execution.** Every node in the cluster executes its assigned aggregations for each phase.

3.2 Estimating the bandwidth

This section describes how GRASP estimates the available bandwidth for data transfers without network topology information. GRASP schedules aggregation plans so that one node sends to and receives from at most one node within a phase to avoid network contention. This ensures that the outgoing link and the incoming link of each node are used by at most one data transfer. Similar approaches are used by Rödiger et al. [41] to minimize network contention.

GRASP measures the pair-wise bandwidth through a benchmarking procedure that is executed on system startup. The bandwidth $B(s \rightarrow t)$ is measured by running a benchmark on every s and t pair individually, where s keeps sending data to t. The average throughput is stored as the estimation of $B(s \rightarrow t)$ in a matrix, where the row index is the sender and the column index is the receiver. (For example, $B(v_0 \rightarrow v_1) = 2$ in Figure 5.) The bandwidth matrix B is computed once and reused for all queries that follow. Section 5.3.1 evaluates the accuracy of the estimation and the robustness of GRASP to estimation errors.

3.3 Estimating the size of intermediate results

GRASP needs to estimate the cardinality of the intermediate result between every node pair s and t for aggregation planning. According to set theory, the size of the union of two sets S and T can be calculated as $|S \cup T| = |S| + |T| - |S \cap T| = \frac{|S|+|T|}{1+J}$, where J is the Jaccard similarity $J = \frac{|S \cap T|}{|S \cup T|}$. Hence one can calculate the cardinality of an aggregation from the cardinality of the input partitions S, T and the Jaccard similarity between them.

Accurately calculating the Jaccard similarity is as expensive as computing the aggregation itself, as it requires collecting both inputs to the same node. GRASP thus estimates the Jaccard similarity using the minhash algorithm [3, 21, 13]. After running minhash, the inputs are represented by a small vector of integers called a *minhash signature*. The minhash signatures are used to estimate the Jaccard similarity between the two sets.

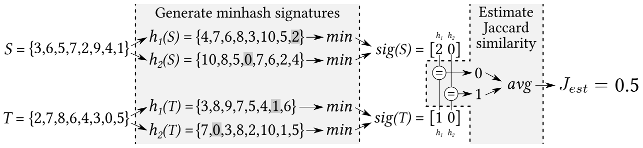

Generate minhash signatures
Estimate Jaccard similarity
$S = \{3,6,5,7,2,9,4,1\}$
$h_1(S) = \{4,7,6,8,3,10,5,2\}$
$h_2(S) = \{10,8,5,0,7,6,2,4\}$
$T = \{2,7,8,6,4,3,0,5\}$
$h_1(T) = \{3,8,9,7,5,4,1,6\}$
$h_2(T) = \{7,0,3,8,2,10,1,5\}$
min
min
min
min
$sig(S) = [2\ 0]$
$sig(T) = [1\ 0]$
h_1 h_2
h_1 h_2
=
=
0
1
avg
$J_{est} = 0.5$

Figure 6: Example of Jaccard similarity estimation with the minhash algorithm and hash functions $h_1(x) = (x + 1)\ mod\ 11$ and $h_2(x) = (3x + 1)\ mod\ 11$.

The minhash algorithm generates minhash signatures by applying a set of hash functions to the dataset. The minhash signature value is the minimum value produced by each hash function. Figure 6 shows an example of the minhash signature calculation for two sets S and T and their minhash signatures $sig(S)$ and $sig(T)$, respectively. The Jaccard similarity between the two sets can be estimated from the minhash signatures as the fraction of the hash functions which produce the same minhash value for both sets. In the example shown in Figure 6, the accurate Jaccard similarity is $J_{acc} = \frac{|S \cap T|}{|S \cup T|} = \frac{6}{10}$. The estimated Jaccard similarity from the minhash signatures is $J_{est} = 1/2$, as only hash function $h_2(\cdot)$ produces the same minhash value between the two sets.

Another appealing property of the minhash algorithm is that the minhash signature $sig(S \cup T)$ can be computed from the minhash signatures $sig(S)$ and $sig(T)$, respectively: The minhash signature of the union is the pairwise minimum of the respective signatures, or $sig(S \cup T)[i] = min\big(sig(S)[i], sig(T)[i]\big)$. The practical significance of this property is that GRASP needs to access the original data only once before the aggregation starts, and then will operate on the much smaller signatures during aggregation planning.

In GRASP, every node partitions the local data and calculates the cardinality and the minhash signatures for each partition. (This is step 2 in Figure 5.) The coordinator collects the cardinality and the minhash signature for each partition of every node in two arrays `Card` and `MinH` of size $|V_C| \times |L|$. The arrays are initialized to $\texttt{Card}[v, l] \leftarrow \big|X_0^l(v)\big|$ and $\texttt{MinH}[v, l] \leftarrow sig\big(X_0^l(v)\big)$. After these arrays are populated with information from every node, they are only accessed by two functions during aggregation planning, which are defined in Algorithm 1. The first function is EstCard(s, t, l) which estimates the Jaccard similarity between the sets $X_i^l(s)$ and $X_i^l(t)$ from their minhash signatures and returns an estimate of the cardinality of their union. The second function is Update(s, t, l) which updates the `Card` and `MinH` arrays after the $s \rightarrow t$ transfer of partition l.

Algorithm 1: EstCard(s,t,l) estimates $\big|X_i^l(s) \cup X_i^l(t)\big|$ and Update(s,t,l) updates the `Card` and `MinH` arrays.

```
Input $s, t \in V_C$: computing node identifiers
      $l \in L$: data partition identifier
function EstCard($s,t,l$)
    sigS $\leftarrow$ MinH$[s, l]$; sigT $\leftarrow$ MinH$[t, l]$; $J \leftarrow 0$
    for $j \in [1, n]$ do
        if sigS$[j]$ = sigT$[j]$ then
            $J \leftarrow J + 1/n$
    return $\frac{\texttt{Card}[s,l] + \texttt{Card}[t,l]}{1 + J}$
function Update($s,t,l$)
    Card$[t, l] \leftarrow$ EstCard($s,t,l$)
    Card$[s, l] \leftarrow 0$
    for $j \in [1, n]$ do
        MinH$[t, l][j] \leftarrow min($MinH$[s, l][j]$, MinH$[t, l][j])$
    MinH$[s, l] \leftarrow \perp$
```

How many hash functions does minhash need? GRASP uses only 100 hash functions so that signatures are less than 1KB. This choice sacrifices accuracy but keeps the computation and network cost small. Satuluri and Parthasarathy [43] show that the estimation is within 10% of the accurate similarity with 95% probability when $n = 100$. Section 5.3.3 evaluates the accuracy of the minhash estimation.

3.4 Forecasting the benefit of each aggregation

Ideally one should take the cost of all future aggregation phases into account when picking the best plan for the current phase. This is prohibitively expensive as there are n^{n-2} possible aggregation trees for a cluster with n nodes [4]. A greedy approach that minimizes the cost of the current phase only ignores how similarity can reduce the network cost of future data transfers. Hence, GRASP looks one phase ahead during optimization to balance the network transfer cost of a data transfer in the current phase with the anticipated future savings from transmitting less data in the next phase.

The heuristic GRASP uses to pick which transfers to schedule in the current phase is based on a cost function $C_i(s, t, l)$ that adds the cost of an $s \to t$ transfer in this phase and the cost of transmitting the union of the data in the next phase. $C_i(s, t, l)$ is constructed based on the following intuition:

1) Penalize the following transfers by setting $C_i = \infty$ so that they will never be picked: (1) Node s sending partitions whose destination is s, to prevent circular transmissions. (2) One node sending a partition to itself, as this is equivalent to a no-op. (3) Transfers involving nodes that neither have any data nor are they the final destination for this partition.

2) When any node transmits partition l to its final destination $M(l)$, only the cost of the data transfer needs to be considered, as this partition will not be re-transmitted again. Hence, we set C_i to $\texttt{COST}(s \to t)$ in this case, where $\texttt{COST}$ is defined in Eq. 5, and $Y_i\,(s \to t) = X^l_{i-1}\,(s)$.

3) Otherwise, add the cost of the $s \to t$ transfer to the cost of transmitting the aggregation result in the next phase. We define $E_i(s, t, l) = \frac{\text{EstCard}(s,t,l) \cdot w}{B(s \to t)}$ to simplify the notation.

Based on the above, we define C_i for a transfer $s \to t$ of partition l between any pair of nodes (s, t) in phase P_i as:

$$C_i(s,t,l) = \begin{cases} \infty & s = t \\ \infty & s = M(l) \\ \infty & X^l_{i-1}\,(s) = \varnothing \\ \infty & X^l_{i-1}\,(t) = \varnothing \\ \texttt{COST}(s \to t) & t = M(l) \\ \texttt{COST}(s \to t) + E_i(s,t,l) & \textit{otherwise} \end{cases} \quad (8)$$

Figure 7 shows C_1 for the phase P_1 of the aggregation shown in Figure 1. There is only one partition in this example, hence $l = 0$. The row index is the sending node and the column index is the receiving node. Note that the matrix C_i will not be symmetric, because transfers $s \to t$ and $t \to s$ transmit different data and use different network links.

3.5 Selecting aggregation pairs

This section describes step 5 in Figure 5 which selects transfers among all possible pairs to produce one aggregation phase P_i. There are three aspects for consideration when selecting candidate aggregations:

$$C_1 = \begin{array}{c|cccc|l} & v_0 & v_1 & v_2 & v_3 & \\ \hline v_0 & \infty & \infty & \infty & \infty & \\ v_1 & 3 & \infty & 9 & 9 & \\ v_2 & 3 & 9 & \infty & \boxed{6} & \dashrightarrow C_1(v_2, v_3, 0) \\ v_3 & 3 & 9 & 6 & \infty & \end{array}$$

Figure 7: The matrix C_1 for the phase P_1 of the aggregation problem in Figure 1 that has a single partition. The example assumes w is equal to the bandwidth B. Rows represent the sender and columns represent the receiver. The circled value corresponds to the aggregation $v_2 \to v_3$ where v_2 sends out $\{D, E, F\}$ to aggregate with $\{D, E, F\}$ in v_3.

1) In each phase, how many transfers does a node participate in? Prior work shows that uncoordinated network communication leads to congestion in the network [41, 42]. Rödiger et al. [41] do application-level scheduling by dividing communication into stages to improve throughput, where in each stage a server has a single target to send to and a single source to receive from. Like prior work, GRASP restricts the communication within one phase to minimize network contention. Specifically, GRASP picks transfers such that one node sends to at most one node and receives from at most one node in each aggregation phase.

2) How many nodes are selected for aggregation in one phase? In order to maximize the network utilization, GRASP picks as many data transfers as possible in one phase until the available bandwidth B is depleted.

3) Given many candidate aggregation pairs, which aggregation should one choose within one phase? GRASP minimizes the C_i function defined in Equation 8 and selects aggregations by picking the smallest C_i values.

Algorithm 2 shows how GRASP selects candidate aggregations for one phase P_i. V_{send} is the set of candidate nodes to be senders, V_{recv} is the set of candidate nodes to be receivers, and V_l is the nodes that can operate on partition l. The algorithm picks the aggregation pair which has smallest value in C_i (line 3). The algorithm then removes the selected nodes from the candidate node sets (lines 6-7) to enforce that (a) one node only sends to or receives from at most one node, and (b) one node does not send and receive data for the same partition within the same phase. Then, the trans-

Algorithm 2: Selecting data transfers for phase P_i

```
   Input $C_i$: the cost function defined in Eq. 8.
         $V_{send}$: candidate nodes to send
         $V_{recv}$: candidate nodes to receive
         $V_l$: candidate nodes to operate on partition $l$
   Output $P_i$: the next aggregation phase
 1 $P_i \leftarrow \varnothing$; $V_{send} \leftarrow V_C$; $V_{recv} \leftarrow V_C$; $V_l \leftarrow V_C$
 2 while $|V_{send}| > 0$ and $|V_{recv}| > 0$ do
 3     Pick $\langle s \to t, l \rangle$ such that
           $s \in (V_{send} \cap V_l)$, $t \in (V_{recv} \cap V_l)$ and
           $C_i(s, t, l)$ has the minimum value in $C_i$
 4     if $C_i(s, t, l) = \infty$ then
 5         break
 6     Remove $s$ from $V_{send}$ and $V_l$, if found
 7     Remove $t$ from $V_{recv}$ and $V_l$, if found
 8     Add $\langle s \to t, l \rangle$ to $P_i$
 9     Update$(s, t, l)$
10 return $P_i$
```

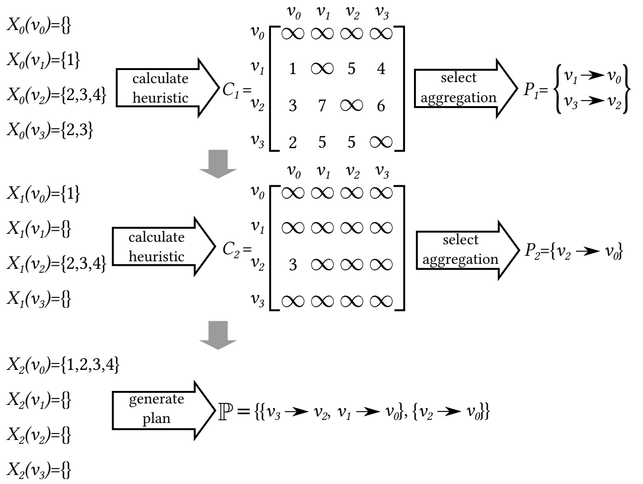


Figure 8: An example of how GRASP generates aggregation plans for an all-to-one aggregation with a single partition.

fer $s \to t$ for partition l is added to the aggregation phase P_i (line 8). GRASP calls the function UPDATE(s, t, l), which was defined in Algorithm 1, to update the minhash signatures and the cardinalities in arrays `MinH` and `Card` (line 9), as data in s and t will change after the aggregation. The algorithm stops when either candidate set is empty (line 2) or there are no more viable transfers in this phase (line 5).

Figure 8 shows an example of how GRASP selects aggregations using the C_i cost function. For simplicity, we again show an all-to-one aggregation with a single partition $l = 0$, and we assume the bandwidth B to be equal to the tuple width w. In the first iteration, the coordinator constructs the matrix C_1 from the cost function described in Section 3.4. For example, assume in the first phase $|X_0(v_2)| = 3$ and $|X_0(v_2) \cup X_0(v_3)| = 3$, then $C_1(v_2, v_3, 0) = 6$. After constructing the cost matrix C_1, GRASP picks data transfers for aggregation using Algorithm 2. The first pick is $v_1 \to v_0$ because it has the least cost. Because a transfer has now been scheduled on the $v_1 \to v_0$ link, GRASP eliminates v_1 and v_0 from all candidate sets. GRASP then picks $v_3 \to v_2$. GRASP then finishes this phase because there are no candidates left, and appends the aggregation phase $P_1 = \{v_1 \to v_0, v_3 \to v_2\}$ to the aggregation plan $\mathbb{P}$. In the next iteration, GRASP constructs matrix C_2 and picks the last data transfer $v_2 \to v_0$ for phase P_2. At this point, all data will have been aggregated to the destination nodes so the aggregation plan $\mathbb{P}$ will be scheduled for execution.

4. HARDNESS OF APPROXIMATION

Many hard problems are amenable to efficient approximation algorithms that quickly find solutions that are within a guaranteed distance to the optimal. For instance, 2-approximation algorithms —polynomial algorithms that return a solution whose cost is at most twice the optimal— are known for many NP-hard minimization problems. A natural question to ask is how closely does GRASP approximate the optimal solution to the aggregation problem.

This section proves that it is not feasible to create a polynomial algorithm that approximates the optimal solution to the aggregation problem within any constant factor. In other words, the aggregation problem is not only NP-hard but it also cannot be approximated within any constant factor by any polynomial algorithm, including GRASP. This hardness of approximation result is much stronger than simply proving the NP-hardness of the problem, as many NP-hard problems are practically solvable using approximation.

The proof is structured as follows. Section 4.1 introduces an assumption regarding the cost of using shared network links. Section 4.2 defines the Small Set Expansion (SSE) problem and the well-established SSE conjecture. Section 4.3 starts with an instance of SSE and reduces it to the all-to-one aggregation problem. This proves that the all-to-one aggregation problem is NP-hard to approximate, assuming the SSE conjecture. Section 4.3.3 proves that the all-to-all aggregation problem is also NP-hard to approximate.

4.1 Link sharing assumption

Whereas GRASP will never schedule concurrent data transfers on the same link in one phase in a star network, the theoretical proof needs a mechanism to assess the runtime cost of sharing a network link for multiple transfers. Our proof makes the fair assumption that the cost of sending data from one node to another is proportional to the total data volume that is transferred over the same link across all aggregations in this phase.

One way to incorporate link sharing information in the cost calculation is to account for the number of concurrent data transfers on the $s \to t$ path when computing the available bandwidth $B(s \to t)$. For example, for the network topology shown in Figure 1 the available bandwidth from s to t, $B(s \to t)$ can be calculated as:

$$B(s \to t) = min\left(\frac{W(\langle s, v_R\rangle)}{d_o(s)}, \frac{W(\langle v_R, t\rangle)}{d_i(t)}\right) \qquad (9)$$

where $W(\langle s, v_R\rangle)$ and $W(\langle v_R, t\rangle)$ are the network bandwidths of the links, $d_o(s)$ denotes the number of data transfers using the $\langle s, v_R\rangle$ link and $d_i(t)$ denotes the number of data transfers using the $\langle v_R, t\rangle$ link in this phase.

4.2 The Small Set Expansion Problem

This subsection defines the Small Set Expansion (SSE) conjecture [37]. We first briefly discuss the intuition behind this problem and we then give a formal definition.

4.2.1 Intuition

A d-regular graph is a graph where each vertex has d edges for some integer $d \geq 1$. The Small Set Expansion problem asks if there exists a small subset of vertices that can be easily disconnected from the rest in a d-regular graph. The SSE conjecture states that it is NP-hard to distinguish between the following two cases: (1) The **YES** case, there exists some small set of vertices that can be disconnected from the graph. (2) The **NO** case, such a set does not exist. In other words, in this case every set of vertices has a relatively large boundary to the other vertices in the graph.

Note that the SSE conjecture is currently open, as it has not been proven or disproven yet. Just like the well-known $\mathbf{P} \neq \mathbf{NP}$ conjecture, the theory community has proceeded to show that many problems are hard to approximate based on the general belief that the SSE conjecture is true. Significant hardness of approximation results that assume the SSE conjecture include the treewidth and pathwidth of a graph [2], the Minimum Linear Arrangement (MLA) and the c-Balanced Separator problem [38].

4.2.2 Formal Definition

Let G be an undirected d-regular graph. For any subset of vertices $S \subseteq V(G)$, we define the *edge expansion* of S to be $\Phi(S) = \frac{E(S, V \setminus S)}{d|S|}$.

Definition 4.1. *Let $\rho \in [-1, 1]$. Let Φ^{-1} be the inverse function of the normal distribution. Let X and Y be jointly normal random variables with mean 0 and covariance matrix $\begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$. We define $\Gamma_\rho \colon [0,1] \to [0,1]$ as $\Gamma_\rho(\mu) = \Pr[X \leq \Phi^{-1}(\mu) \wedge Y \leq \Phi^{-1}(\mu)]$.*

Conjecture 4.2 (The Small Set Expansion conjecture [37]). *For every integer $q > 0$ and $\varepsilon, \gamma > 0$, it is NP-hard to distinguish between the following two cases:*

YES *There is a partition of $V(G)$ into q equi-sized sets $S_1, \ldots, S_q$ such that $\Phi(S_i) \leq 2\varepsilon$, $\forall i \in \{1, \ldots, q\}$.*

NO *For every $S \subseteq V(G)$ we have $\Phi(S) \geq 1 - (\Gamma_{1-\varepsilon/2}(\mu) + \gamma)/\mu$, where $\mu = |S|/|V(G)|$.*

Remark 4.1. *In the **YES** case, the total number of edges that are not contained in one of the S_i sets is at most $2\varepsilon|E|$.*

Remark 4.2. *In the **NO** case, for every $S \subseteq V(G)$ with $|V(G)|/10 \leq |S| \leq 9|V(G)|/10$, we have $|E(S, V(G) \setminus S)| \geq c\sqrt{\varepsilon}|E(G)|$, for some constant $c > 0$.*

4.3 Hardness of the aggregation problem

Before stating the formal inapproximability result, we first provide the intuition behind our proof strategy approach. We then reduce the SSE problem to the all-to-one aggregation problem. Finally, we show that the all-to-all problem is a straightforward generalization of the all-to-one problem.

4.3.1 Intuition

We now give a brief intuitive overview of the proof. Recall that in the SSE problem we are given a graph G and the goal is to decide whether G admits a partition into small subgraphs, each having a small boundary (a SSE partition henceforth), or G is an expander at small scales; that is, all small subgraphs of G have a large boundary. The SSE conjecture asserts that this problem is hard to approximate, and has been used to show the inapproximability of various graph optimization problems [2]. Inspired by these results, we show that the all-to-one aggregation problem is hard to approximate by reducing the SSE problem to it. Our proof strategy is as follows. We begin with an instance G' of the SSE problem. We encode G' as an instance of the all-to-one aggregation problem by interpreting each node of G' as a leaf node in the star network, and each edge $\langle u, v\rangle$ of G as a data item which is replicated in nodes u and v in the aggregation problem. We show that any partition of G can be turned into an aggregation protocol, and, conversely, any aggregation protocol can be turned into a partition of G. The key intuition is that the cost of the partition is related to the cost of the aggregation via the observation that the data items that need to be transmitted twice are exactly the edges that are cut by the partition.

4.3.2 Formal proof for the all-to-one aggregation

Suppose that we are given an all-to-one aggregation instance: a graph G, a single destination vertex $v^* \in V(G)$, and the data $X_0(v)$ in each node $v \in V(G)$. Let $X = \bigcup_{v \in V(G)} X_0(v)$ be the set of all data. Let $\mathbb{P} = \{P_1, P_2, \ldots, P_n\}$ be an execution plan. For every $P_i = \{s_1 \to t_1, \ldots, s_k \to t_k\} \in \mathbb{P}$, let $S(P_i) = \{s_1, \ldots, s_k\}$ and $T(P_i) = \{t_1, \ldots, t_k\}$.

We define the *overhead cost* of $\mathbb{P}$ to be $\texttt{COST}\,(\mathbb{P}) - |X|$. Under the all-to-one aggregation model, every execution plan is obtained from an aggregation tree. To simplify the proof, we assume that one node sends data to only one node within a phase. This modeling assumption is acceptable from a theoretical standpoint as one can represent a phase where a node transmits data to multiple destinations as a sequence of discrete phases to each individual destination. We say that $\mathbb{P}$ is obtained from an *aggregation tree T_P*, if the following conditions hold:

1. T_P is a spanning tree of G, rooted at v^*.
2. The leaf vertices of T_P are exactly the elements of $S(P_1)$. Furthermore, for every $i \in \{2, \ldots, k-1\}$, the leaf vertices of $T_P \setminus \bigcup_{1 \leq j < i} S(P_j)$ are exactly the elements of $S(P_i)$.

Theorem 4.3. *For every $\varepsilon > 0$, given an aggregation instance $\big(G, v^* \in V(G), X_0(v)\ \forall v \in V(G)\big)$, it is SSE-hard to distinguish between the following two cases:*

YES *There exists an execution plan that is obtained from an aggregation tree with overhead cost $O(\varepsilon|X|)$.*

NO *Every execution plan that is obtained from an aggregation tree has overhead cost $\Omega(\sqrt{\varepsilon}|X|)$.*

Proof. We start with an instance of SSE with $q = 1/\varepsilon$, and reduce it to our problem. Let G' be the d-regular graph of the SSE instance. We construct an aggregation instance as follows. Let $V(G) = V(G')$, and $X = E(G')$. Note that G is a complete graph with the same vertex set as G'. For every $v \in V(G)$, let $X_0(v) = \{\langle u, w\rangle \in X : v = u \vee v = w\}$ be the set of data that is held by v.

In the **YES** case of the SSE instance, we have disjoint sets $S_1, S_2, \ldots, S_q$ of equal size. For every $i \in \{1, 2, \ldots, q\}$, we have $|S_i| = |V(G)|/q = \varepsilon|V(G)|$. We may assume w.l.o.g. that $v^* \in S_q$. For every $i \in \{1, 2, \ldots, q-1\}$, pick an arbitrary vertex $v_i \in S_i$. Let also $v_q = v^*$. For every $j \in \{1, 2, \ldots, q\}$, let $\{s_{i,1}, \ldots, s_{i,i_j}\} = S_j \setminus \{v_j\}$. We first construct an aggregation tree T as follows. For every $i \in \{1, 2, \ldots, q\}$, let v_i be the parent of all other vertices in S_i. Let v_q be also the parent of $v_1, v_2, \ldots, v_{q-1}$.

Now consider the execution plan corresponding to T. This aggregation has two phases: $\mathbb{P} = \{P_1, P_2\}$. First we describe P_1. For each S_i, we aggregate all the data held by vertices of S_i to v_i; that is every vertex in S_i (except v_i itself) transfers its dataset to v_i. This can be done simultaneously for all S_i's, since S_i's are disjoint sets. We have that $P_1 = \{s_{1,1} \to v_1,\ s_{1,2} \to v_1,\ \ldots,\ s_{1,i_1} \to v_1,\ s_{1,2} \to v_2,\ \ldots,\ s_{1,i_2} \to v_2,\ \ldots,\ s_{q,1} \to v_q,\ \ldots,\ s_{q,i_q} \to v_q\}$.

By the construction, at the beginning for each vertex v we have that $|X_0(v)| = d$. Therefore, for every S_i, the total volume of data to be transferred to v_i is $2\varepsilon|E(G)| = d\varepsilon|V(G)| = d|S_i|$. In other words, for every $(s_{i,j} \to v_i) \in P_1$, we have that $\texttt{COST}(s_{i,j} \to v_i) = 2\varepsilon|E(G)|$, and thus we have $\texttt{COST}(P_1) = 2\varepsilon|E(G)|$.

In the second phase of the execution plan, for every $i \in \{1, 2, \ldots, q-1\}$, we need to transfer all the data held by v_i to v^*. This can be done simply by sending one data at a time to v^*. We have:

$$P_2 = \{v_1 \to v_q, v_2 \to v_q, \ldots, v_{q-1} \to v_q\}$$

By Remark 4.1, the total number of tuples that are transferred more than once in this phase is at most $\varepsilon d|V(G)| = 2\varepsilon|E(G)|$. This means that $\texttt{COST}(P_2) \leq (1+2\varepsilon)|E(G)|$. Therefore we have that $\texttt{COST}(\mathbb{P}) \leq (1+4\varepsilon)|E(G)|$, and thus the overhead cost of this execution plan is $O(\varepsilon|E(G)|)$.

In the **NO** case, we want to show that every execution plan that is obtained from an aggregation tree has cost $\Omega(\sqrt{\varepsilon}|E|)$. Let $\mathbb{P}$ be an execution plan that is obtained from an aggregation tree T. For every $v \in V(T)$, let T_v be the subtree of T rooted at v.

Suppose that v^* has a child v such that $|V(T)|/10 \leq |V(T_v)| \leq 9|V(T)|/10$. We apply Remark 4.2 by setting $S = T_v$. We have that $E(S, V(G)\backslash S) \geq c\sqrt{\varepsilon}|E(G)|$, for some constant $c > 0$. This means that there are at least $c\sqrt{\varepsilon}|E(G)|$ data that are going to be sent at least twice to v^* in the execution plan, or $\texttt{COST}(\mathbb{P}) = \Omega((1+\sqrt{\varepsilon})|E(G)|)$. Thus, the overhead cost of this execution plan is $\Omega(\sqrt{\varepsilon}|E(G)|)$.

Otherwise, v^* has a child v such that $|V(T_v)| < |V(T)|/10$. In this case, there are at least $9|E(G)|/10$ data in T_v that are going to be transferred at least twice to get to v^* in the execution plan. Therefore, we have $\texttt{COST}(\mathbb{P}) = \Omega((0.9 + 0.9)|E(G)|)$, and thus the overhead cost of this execution plan is clearly $\Omega(\sqrt{\varepsilon}|E(G)|)$. This completes the proof. □

Corollary 4.4. *Assuming Conjecture 4.2, it is NP-hard to approximate the minimum overhead cost of an all-to-one aggregation plan that is obtained from an aggregation tree within any constant factor.*

Corollary 4.5. *Assuming Conjecture 4.2, it is NP-hard to find an all-to-one aggregation plan that is obtained from an aggregation tree with minimum cost.*

One might ask if it is feasible to brute-force the problem for small graphs by enumerating all possible aggregation trees and picking the best solution. Unfortunately this would be extremely expensive even for small graphs. Cayley's formula [4] states that the number of different spanning trees of graph with n vertices is n^{n-2}. Hence, even for $n = 20$ one needs to enumerate $20^{18} \geq 10^{23}$ different trees.

4.3.3 Formal proof for the all-to-all aggregation

The more general case is the all-to-all aggregation problem. We observe that the all-to-one aggregation problem can be trivially reduced to the all-to-all aggregation problem, since by the definition, every instance of the all-to-one aggregation problem is also an instance of the all-to-all aggregation problem.

Theorem 4.6. *Assuming Conjecture 4.2, it is NP-hard to find an all-to-all aggregation plan with minimum cost.*

Proof. We reduce the all-to-one aggregation problem to the all-to-all aggregation problem. Suppose that we are given an instance of the all-to-one aggregation problem. By its definition, this is also an instance of the all-to-all aggregation problem where the mapping M is such that the aggregation destination of every partition is node $v^* \in V_C$. By Corollary 4.5 we know that the all-to-one aggregation problem is NP-hard assuming Conjecture 4.2, therefore the all-to-all aggregation problem is NP-hard as well. □

5. EXPERIMENTAL EVALUATION

This section compares the GRASP algorithm with the repartition algorithm and LOOM. Section 5.1 introduces the experimental setup, which includes the hardware setting, the workloads and the baselines. The other sections evaluate the following questions:

- (§ 5.2.1) How well does GRASP leverage similarity between datasets?
- (§ 5.2.2) Can GRASP benefit from workload imbalance?
- (§ 5.3.1) How accurate is the bandwidth estimation? How robust is GRASP to estimation errors?
- (§ 5.3.2) How does GRASP perform in nonuniform networks?
- (§ 5.3.3) Is GRASP faster than aggregation based on repartitioning and LOOM on TPC-H and real datasets?
- (§ 5.3.4) How well does GRASP work in a real-world deployment where the network conditions are unpredictable?

The evaluation of how GRASP utilizes the similarity within the dataset and how GRASP scales out is included in an extended version of this paper [25].

5.1 Experimental setup

We implemented the GRASP framework in C++ and we have open-sourced our prototype implementation [15]. We evaluate GRASP in two clusters. The first is a shared cluster connected by a 1 Gbps network. Each machine has two NUMA nodes with two Intel Xeon E5-2680v4 14-core processors and 512 GB of memory. The second cluster is Amazon EC2 with d2.8xlarge instances which have 36 vCPUs and 244 GB of memory. The instances are connected with a 10 Gbps network.

We run one or more aggregation fragments in each machine/instance. Hence, one fragment corresponds to one logical graph node in Figure 1. We evaluate all-to-all aggregations by setting the mapping between partitions and destinations so that aggregation results are evenly balanced across all nodes. We evaluate all-to-one aggregations by mapping all data partitions to the same destination.

Our evaluation reports the total response time to complete the aggregation query. All our performance results include the time to plan the aggregation using GRASP, the time to transfer all data to their destinations and the time to process the aggregation locally in each node. All experiments use hash-based local aggregation.

5.1.1 Baselines

We compare GRASP with two baselines. The first baseline is LOOM [8, 9]. As described in Section 1, LOOM needs the size of aggregation results during query planning. In our evaluation we configure LOOM to use the accurate result size so that LOOM achieves its best performance. The second baseline is repartitioning which has two versions. One version is without local aggregation, where data is directly sent to the destination fragment for aggregation. We use "Repart" to denote this version. The other version is with local aggregation, where data is first aggregated locally, then the local aggregation result is sent to the destination fragment for aggregation. We use "Preagg+Repart" to denote this version of repartitioning. Note that repartitioning works for both all-to-all and all-to-one aggregations, while LOOM only works for all-to-one aggregations.

5.1.2 Workloads

We use five workloads in our evaluation.

1) Synthetic workload. The first workload is a synthetic workload which has one table `R`, with two long integers `R.a` and `R.b` as attributes. The query evaluated is `SELECT R.a SUM(R.b) FROM R GROUP BY R.a`.

2) TPC-H workload. The second workload is the TPC-H workload with scale factor 80. We evaluate this subquery from TPC-H Q18: `SELECT ORDERKEY, SUM(QUANTITY) FROM LINEITEM GROUP BY ORDERKEY`. The `LINEITEM` table is partitioned and distributed on the `SUPPKEY` to framgents with a modulo hash function.

3) MODIS workload. The third workload is the Surface Reflectance data MOD09 from MODIS (Moderate Resolution Image Spectroradiometer) [46]. The MODIS data provides the surface relfectance of 16 bands together with the location coordinates (latitude and longitude). In the processing of MODIS data, one product is MOD09A1 [47] which aggregates the observed data in an 8-day period with the following query: `SELECT Latitude, Longitude, MIN(Band3) FROM RelfectTable GROUP BY ROUND(Latitude, 2), ROUND(Longitude, 2) WHERE Date BETWEEN '01/01/2017' AND '01/08/2017'`. The MODIS data is stored in separate files, one file per satelite image in timestamp order. We download about 1200 files from the MODIS website, and assigned files into plan fragments in a round-robin fashion. Overall, there are about 3 billion tuples and 648 million distinct GROUP BY keys in this dataset.

4) Amazon workload. The fourth dataset is the Amazon review dataset [19]. The review dataset has more than 82 million reviews from about 21 million users. The dataset includes the reviewer ID, overall rating, review time and detail review etc. We evaluate the following query to calculate the average rating a customer gives out. `SELECT ReviewerID, AVG(OverallRate) FROM AmazonReview GROUP BY ReviewerID`. The reviews are stored in timestamp order and we split this file into plan fragments.

5) Yelp workload. The fifth dataset is the Yelp review dataset [53]. The review dataset has more than 5 million reviews from about 1.3 million users. The Yelp dataset has similar attributes as the Amazon dataset and we use a similar query to calculate the average stars a customer gives.

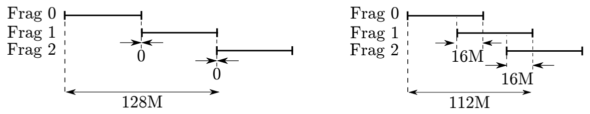
5.2 Experiments with uniform bandwidth

This section evaluates GRASP in a setting where each plan fragment communicates with the same bandwidth. The measured inter-fragment bandwidth is 118 MB/s. We experiment with 8 machines and 1 fragment per machine, which results in 8 fragments in total. We use the synthetic workload in this section.

5.2.1 Effect of similarity across fragments

GRASP takes advantage of the similarities between datasets in different fragments in aggregation scheduling. How well does the GRASP algorithm take advantage of similarities between datasets?

In this experiment, we change the similarities between datasets, i.e. the number of common GROUP BY keys, in different plan fragments. Each plan fragment has 64 million tuples. Figure 9 shows how we change the similarity between datasets. Each segment in Figure 9 shows the range of $R.a$ in one fragment. Figure 9 only shows fragments 0, 1 and 2. The range of datasets between adjacent fragments has an overlap. The Jaccard similarity increases when the size of the overlap increases.



(a) Jaccard similarity $J = \frac{0}{128}$. (b) Jaccard similarity $J = \frac{16}{112}$.

Figure 9: The line segments represent the range of GROUP BY attributes. The Jaccard similarity increases when the overlap of GROUP BY key ranges increases.

The experimental results for all-to-one aggregation are shown in Figure 10. The horizontal axis is the Jaccard similarity coefficient between datasets. The vertical axis is the speedup over the Preagg+Repart algorithm with Jaccard similarity 0. Here speedup 1 corresponds to response time of 64.6 seconds. Figure 10 shows that GRASP has the best performance and is up to 4.1× faster than Preagg+Repart and 2.2× faster than LOOM when the Jaccard similarity is 1. Figure 10 shows that the performance of Repart and Preagg+Repart stays the same when the Jaccard similarity changes. This means that repartitioning cannot utilize the similarities between datasets.

GRASP has better performance than LOOM for two reasons. First, GRASP is data distribution-aware and prioritizes aggregations with higher similarity. Second, GRASP has higher network utilization than LOOM. In GRASP, a fragment can be both sending and receiving as long as it is not working on the same partition. In LOOM, a fragment is either a parent fragment receiving data or a child fragment sending data. We also evaluate with all-to-all aggregation. The result shows that GRASP has similar performance with repartitioning as there is no underutilization of the network during the all-to-all aggregation. We omit the results for brevity.

5.2.2 Effect of workload imbalance

In parallel aggregation, some fragments may receive more tuples to aggregate for two reasons. First, the repartition function may assign more GROUP BY keys to some fragments. Second, even if each fragment gets the same number of GROUP BY keys to process, there may be skew in the dataset. In this section, we evaluate how GRASP works when one fragment gets more tuples to process.

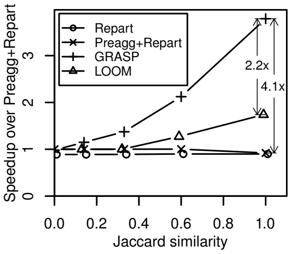


Figure 10: Speedup of GRASP when the similarity between datasets increases. GRASP is up to 2.2× faster than LOOM and 4.1× faster than Preagg+Repart.

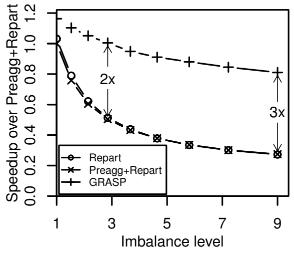


Figure 11: Speedup of GRASP for all-to-all aggregations when the fragment 0 receives more tuples. GRASP is up to 3× faster than Preagg+Repart.

In this experiment, we have 128 million tuples and $R.a$ ranges from 1 to 128 million. We change the repartition function to assign more tuples to fragment 0. We assign n million tuples to fragment 0 for aggregation and assign $m = \frac{128-n}{7}$ million tuples to the other fragments. We use $l = \frac{n}{m}$ to denote the *imbalance level*. When n equals to 16, l is 1 and there is no imbalance. However, as n increases, fragment 0 gets more tuples than other fragments.

The results are shown in Figure 11. The horizontal axis is *imbalance level* l. The vertical axis is the speedup over Preagg+Repart when l is 0. Here speedup 1 corresponds to response time of 22.1 seconds. Notice that LOOM is not shown here because LOOM does not work for all-to-all aggregations. Figure 11 shows that the performance of repartition and GRASP both decreases when the workload imbalance increases. However, the performance decreases much faster for repartition than GRASP and GRASP is already 2× faster than Preagg+Repart when fragment 0 receives about 3 times of data of other fragments. This is because in repartition, other fragments will stop receiving and aggregating data when they are waiting for fragment 0 to complete. While for GRASP, other fragments are still scheduled to receive and aggregate data. GRASP improves performance when some fragments process more tuples.

5.3 Experiments with nonuniform bandwidth

GRASP is cognizant of the network topology, which is crucial when the communication bandwidth is nonuniform, i.e. when some plan fragments communicate at different speeds. The distribution of the link bandwidth is not uniform in many common network topologies. Datacenter networks often have large oversubscription ratios and data transfers within the same rack will be faster than data transfers across racks [17]. The data transfer throughput between instances in the cloud is also nonuniform [27]. Even HPC systems which strive for balanced networks may have nonuniform configurations [20].

This section evaluates how GRASP performs when the network bandwidth is nonuniform. All experiments in this section run multiple concurrent plan fragments in each server to emulate a nonuniform network where some data transfers will be faster than others due to locality.

5.3.1 Impact of bandwidth estimation

The bandwidth estimation procedure described in Section 3.2 leads to two questions: how accurate is the estimation and how robust is GRASP to estimation errors?

Figure 12 compares the available bandwidth as estimated by GRASP versus a manual calculation based on the hardware specifications, the network topology and the fragment placement. This experiment uses 8 machines with each machine having 14 fragments in the experiment. “Within machine” and “Across machines” corresponds to the communication bandwidth between fragments within the same node and across different nodes, respectively. The result shows that the estimation error is within 20% from the theoretical bandwidth. We conclude that the GRASP estimation procedure is fairly accurate in an idle cluster.

The estimation procedure may introduce errors in production clusters that are rarely idle. Figure 13 shows the impact of bandwidth underestimation on the response time of the aggregation plan produced by GRASP. We test two underestimation levels, 20% and 50% from the theoretical value. In this experiment we force GRASP to use a modified bandwidth matrix while running the aggregation query on the MODIS dataset. We run the experiment 10 times picking nodes at random for each setting, and show the standard deviation as an error bar. Co-location results in the underestimation of the communication bandwidth between local fragments in one or more machines. NIC contention and switch contention underestimates the available network bandwidth for one or all nodes in the cluster, respectively. “Topology” corresponds to the calculation based on the hardware capabilities, while “GRASP estimation” corresponds to the procedure described in Section 3.2. The horizontal axis is the response time difference with respect to the plan GRASP generated using the theoretical hardware capabilities (hence, lower means faster). The result shows that GRASP has better performance when using the estimated bandwidth matrix than the accurate bandwidth from network topology. This is because the estimated bandwidth measured from the benchmark is closer to the available bandwidth during query execution. Moreover, even when the available bandwidth is underestimated by up to 50%, the change in query response time is less than 20%. We conclude that GRASP is robust to errors introduced during bandwidth approximation.

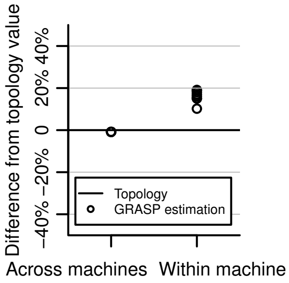


Figure 12: Comparing between the theoretical bandwidth and the bandwidth estimated from benchmarks.

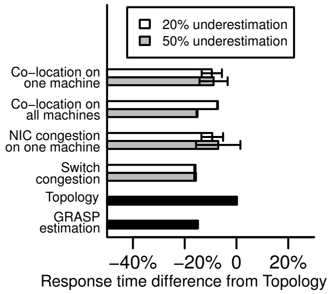


Figure 13: Speedup on the MODIS dataset when changing the estimated bandwidth.

5.3.2 Effect of nonuniform bandwidth

GRASP takes network bandwidth into consideration in aggregation scheduling. How well does GRASP work when the bandwidth between network links is different in a cluster?

In this experiment, we use 4 machines and each machine has 14 aggregation fragments. The dataset in each fragment has 14 million tuples with $R.a$ ranging from 1 to 14 million.

The result is shown in Figure 14. The vertical axis is the speedup over Preagg+Repart. The results show that GRASP has better performance than both repartitioning and LOOM in both all-to-one and all-to-all aggregations. GRASP is up to 16× faster than Preagg+Repart and 5.6× faster than LOOM in all-to-one aggregation and 4.6× faster than Preagg+Repart in all-to-all aggregation. This is because GRASP is topology-aware and schedules more aggregations on the faster network links. GRASP is topology-aware and has better performance than the baselines when the bandwidth between fragments is not uniform.

5.3.3 Real datasets and the TPC-H workload

These experiments evaluate the performance of the GRASP plans with the TPC-H workload and three real datasets. We

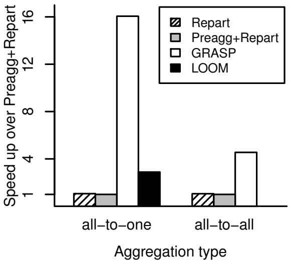


Figure 14: Speedup over Preagg+Repart with nonuniform bandwidth.

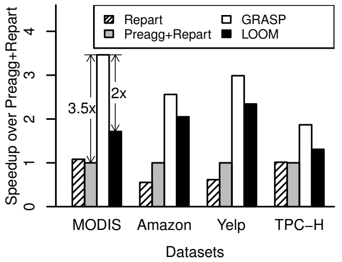
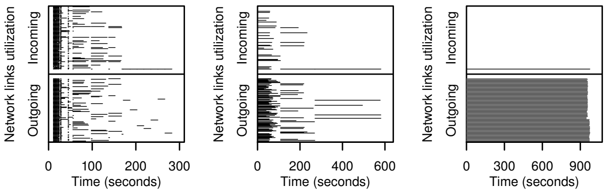


Figure 15: Speedup over Preagg+Repart on the TPC-H workload and on real datasets.



(a) GRASP (b) LOOM (c) Preagg+repart

Figure 16: Network link utilization.

use 8 machines and 14 fragments per machine. The dataset is aggregated to fragment 0, which corresponds to the all-to-one aggregation.

Speedup results: Figure 15 shows the speedup over Preagg +Repart for each algorithm. The result shows that GRASP has the best performance for all datasets. GRASP is 2× faster than LOOM and 3.5× faster than Preagg+Repart in the MODIS dataset.

Network utilization: Figure 16 shows the network utilization plot for the MODIS dataset. The horizontal axis is the time elapsed since the query was submitted to the coordinator. (Note that the scale of the horizontal axis is not the same, as some algorithms finish earlier than others.) Each horizontal line in the plot represents one incoming network link or one outgoing link of a fragment. For each link, we plot a line when there is traffic in the link and leave it blank otherwise.

Figure 16a shows network utilization with GRASP. After a short delay to compute the aggregation plan, the network is fully utilized in the first few phases and there is traffic in all links. As the aggregation progresses, more fragments contain no data and hence these fragments do not further participate in the aggregation. The aggregation finishes in under 300 seconds.

Figure 16b shows LOOM. One can see that the network links, especially the receiving links, are not as fully utilized as in Figure 16a. The fan-in of the aggregation tree produced by LOOM is 5 for this experiment, which makes the receiving link of the parent fragment to be bottleneck. The aggregation finishes in about 600 seconds.

Figure 16c shows Preagg+Repart. All receiving links except fragment 0 (the aggregation destination) are not utilized. The entire aggregation is bottlenecked on the receiving capability of fragment 0. The aggregation takes more than 900 seconds. We omit the figure for Repart as it is similar to Preagg+Repart.

Table 2: Tuples received by the final destination fragment.

Repart	Preagg+Repart	LOOM	GRASP
3,464,926,620	3,195,388,849	2,138,236,114	787,105,152

Tuples transmitted to destination: The GRASP performance gains can be directly attributed to the fact that it transmits less data on the incoming link of the destination fragment, which is frequently the bottleneck of the entire aggregation. Table 2 shows how many tuples the destination fragment receives under different algorithms. Local pre-aggregation has minimal impact as it is only effective when duplicate keys happen to be co-located on the same node. LOOM transmits fewer tuples to the destination fragment as tuples are combined in the aggregation tree before arriving at the final destination fragment. By aggressively combining fragments based on their similarity, GRASP transmits 2.7× less tuples than LOOM to the destination fragment.

Accuracy of minhash estimation: We also evaluate the accuracy of the minhash estimation with the MODIS dataset. Figure 17 shows the cumulative distribution function of the absolute error in estimating the size of the intersection between fragments when the cardinality of the input is accurately known. The result shows that the absolute error of the size of the intersection is less than 10% for 90% of the estimations. We conclude that the minhash estimation is accurate and it allows GRASP to pick suitable fragment pairs for aggregation.

5.3.4 Evaluation on Amazon EC2

This section evaluates GRASP on the MODIS dataset on Amazon EC2. We allocate 8 instances of type d2.8xlarge and run 6 fragments in each instance. Figure 18 shows the speedup over the Preagg+Repart algorithm for each algorithm. Preagg+Repart has better performance than Repart in this experiment. This is because the fast 10 Gbps network in EC2 makes the query compute bound. The throughput of the local aggregation on pre-aggregated data is measured to be 811 MB/s, which is faster than aggregation on raw data with throughput to be 309 MB/s. This does not make a difference in the experiment in Section 5.3.3, as aggregation is network bound in the 1 Gbps network where the maximum throughput is 125 MB/s. However, the aggregation is compute bound in the 10 Gbps network of EC2 with a maximum throughput of 1.2 GB/s, hence pre-aggregation makes a big difference.

Figure 18 shows that GRASP is 2.2× faster than Preagg+ Repart and 1.5× faster than LOOM. GRASP still has bet-

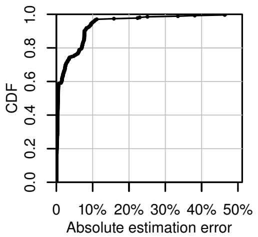


Figure 17: Absolute error in minhash estimation.

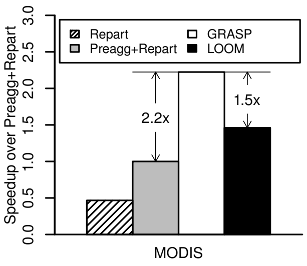


Figure 18: Speedup over Preagg+Repart on the MODIS dataset in Amazon EC2.

ter performance when computation is the bottleneck. This is because GRASP maximizes network utilization by scheduling as many aggregations as possible in each phase, which also maximizes the number of fragments participating in the aggregation and sharing the computation load of each phase.

6. RELATED WORK

Aggregation execution

Aggregation has been extensively studied in previous works. Many works have focused on how to execute an aggregation efficiently in a single server. Larson [23] studied how to use partial aggregation to reduce the input size of other operations. Cieslewicz and Ross [6] evaluated aggregation algorithms with independent and shared hash tables on multicore processors. Ye et al. [52] compared different in-memory parallel aggregation algorithms on the Intel Nehalem architecture. Raman et al. [39] described the grouping and aggregation algorithm used in DB2 BLU. Müller et al. [34] proposed an adaptive algorithm which combines the hashing and sorting implementations. Wang et al. [48] proposed a NUMA-aware aggregation algorithm. Jiang and Gagan [22] and Polychroniou et al [35] used SIMD and MIMD to parallelize the execution of aggregation. Gan et al. [12] optimized high cardinality aggregation queries with moment based summaries. Müller et al. [33] studied the floating-point aggregation.

Aggregation has also been studied in the parallel database system literature. Graefe [14] introduced aggregation evaluation techniques in parallel database system. Shatdal and Naughton [45] proposed adaptive algorithms which switch between the repartition and the two-phase algorithm at runtime. Aggregation trees are used in accelerating parallel aggregations. Melnik et al. [32] introduced Dremel, which uses a multi-level serving tree to execute aggregation queries. Yuan et al. [54] compared the interfaces and implementations for user-defined distributed aggregations in several distributed computing systems. Mai et al. [31] implemented NetAgg which aggregates data along network paths. Costa et al. [7] proposed Camdoop, which does in-network aggregation for a MapReduce-like system in a cluster with a direct-connect network topology. Yalagandula and Dahlin [51] designed a distributed information management system to do hierarchical aggregation in networked systems. Culhane et al. [8, 9] proposed LOOM, which builds an aggregation tree with fixed fan-in for all-to-one aggregations.

The impact of the network topology on aggregation has been studied. Gupta et al. [18] proposed an aggregation algorithm that works in unreliable networks such as sensor networks. Madden et al. [29] designed an acquisitional query processor for sensor networks to reduce power in query evaluation. Madden et al. [28, 30] also proposed a tiny aggregation service which does in network aggregation in sensor networks. Chowdhury et al. [5] proposed Orchestra to manage network activities in MapReduce systems.

None of the above aggregation algorithms is cognizant of the similarity between datasets as GRASP is. The most relevant work is LOOM which considers the amount of data reduction in an aggregation during planning. However LOOM only considers the overall reduction rate and does not consider data similarities during aggregation. The biggest strength of GRASP is that it carefully estimates the size of every partial aggregation and handles each partition differently, which is not possible with LOOM.

Distribution-aware algorithms

Distribution-aware algorithms use information about the distribution and the placement of the data during query processing. Prior works have extensively studied how to take advantage of locality. Some algorithms consider the offline setting. Zamanian et al. [55] introduced a data partitioning algorithm to maximize locality in the data distribution. Prior works have also considered how to extract and exploit locality information at runtime. Rödiger et al. [42] proposed a locality-sensitive join algorithm which first builds a histogram for the workload, then schedules the join execution to reduce network traffic. Polychroniou [36] proposed track-join, where the distribution of the join key is exchanged across the cluster to generate a join schedule to leverage locality. Lu et al. [26] proposed AdaptDB, which refines data partitioning according to access patterns at runtime.

Distribution-aware algorithms have also been proposed to deal with skewed datasets. DeWitt et al. [10] handled skew in a join by first sampling the data, then partitioning the build relation and replicating the probe relation as needed. Shah et al. [44] implemented an adaptive partitioning operator to collect dataset information at runtime and address the problem of workload imbalance in continuous query systems. Xu et al. [50] addressed skew in parallel joins by first scanning the dataset to identify the skewed values, then keeping the skewed rows locally and duplicating the matching rows. Rödiger et al. [40] adopted similar approach as DeWitt et al. [10] by first sampling 1% of the data and then use this information to decide the data partition scheme. Wolf et al. [49] divided the parallel hash join into two phases, and add one scheduling phase to split the partition with data skew. Elseidy et al. [11] proposed a parallel online dataflow join which is resilient to data skew.

7. CONCLUSIONS AND FUTURE WORK

Parallel aggregation is a ubiquitous operation in data analytics. For low-cardinality parallel aggregations, the network cost is negligible after the data has been aggregated locally using pre-aggregation. However, the network communication cost becomes significant for high-cardinality parallel aggregations. This paper proposes GRASP, an algorithm that schedules parallel aggregation in a distribution-aware manner to increase network utilization and reduce the communication cost for algebraic aggregations.

Looking ahead, GRASP can be further extended in two promising ways. First, GRASP can be extended for non-algebraic aggregations. This would require a new metric to quantify the data reduction of an aggregation pair. Second, the assumption that the communication cost dominates the aggregation marginally holds on 10 Gbps networks, and will not hold in faster networks such as InfiniBand. One opportunity is to augment the cost estimation formulas to account for compute overheads, instead of modeling the network transfer cost alone. This can jointly optimize compute and communication overheads during aggregation in high-performance networks.

Acknowledgements: We would like to acknowledge Srinivasan Parthasarathy, Jiongqian Liang, Vishal Dey and the anonymous reviewers for their insightful comments that improved this paper. This work was supported by the National Science Foundation grants IIS-1464381, CCF-1816577, CCF-1815145, CCF-1423230 and CAREER award 1453472.

8. REFERENCES

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, Aug. 2008.

[2] P. Austrin, T. Pitassi, and Y. Wu. Inapproximability of Treewidth, One-shot Pebbling, and Related Layout Problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 13–24. Springer, 2012.

[3] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise Independent Permutations (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 327–336, New York, NY, USA, 1998. ACM.

[4] A. Cayley. A theorem on trees. *Quarterly Journal of Pure Applied Mathematics*, 23:376–378, 1889.

[5] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing Data Transfers in Computer Clusters with Orchestra. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 98–109, New York, NY, USA, 2011. ACM.

[6] J. Cieslewicz and K. A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 339–350. VLDB Endowment, 2007.

[7] P. Costa, A. Donnelly, A. I. T. Rowstron, and G. O'Shea. Camdoop: Exploiting In-network Aggregation for Big Data Applications. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 29–42, 2012.

[8] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. LOOM: Optimal Aggregation Overlays for In-memory Big Data Processing. In *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'14, pages 13–13, Berkeley, CA, USA, 2014. USENIX Association.

[9] W. Culhane, K. Kogan, C. Jayalath, and P. Eugster. Optimal communication structures for big data aggregation. In *2015 IEEE Conference on Computer Communications, INFOCOM 2015, Kowloon, Hong Kong, April 26 - May 1, 2015*, pages 1643–1651, 2015.

[10] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th International Conference on Very Large Data Bases*, VLDB '92, pages 27–40, San Francisco, CA, USA, 1992. Morgan Kaufmann Publishers Inc.

[11] M. Elseidy, A. Elguindy, A. Vitorovic, and C. Koch. Scalable and Adaptive Online Joins. *PVLDB*, 7(6):441–452, 2014.

[12] E. Gan, J. Ding, K. S. Tai, V. Sharan, and P. Bailis. Moment-Based Quantile Sketches for Efficient High Cardinality Aggregation Queries. *CoRR*, abs/1803.01969, 2018.

[13] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[14] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[15] GRASP. `https://code.osu.edu/pythia/grasp`.

[16] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *Proceedings of the Twelfth International Conference on Data Engineering, February 26 - March 1, 1996, New Orleans, Louisiana*, pages 152–159, 1996.

[17] A. G. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Barcelona, Spain, August 16-21, 2009*, pages 51–62, 2009.

[18] I. Gupta, R. v. Renesse, and K. P. Birman. Scalable Fault-Tolerant Aggregation in Large Process Groups. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[19] R. He and J. McAuley. Ups and Downs: Modeling the Visual Evolution of Fashion Trends with One-Class Collaborative Filtering. In *Proceedings of the 25th International Conference on World Wide Web, WWW 2016, Montreal, Canada, April 11 - 15, 2016*, pages 507–517, 2016.

[20] `https://htor.inf.ethz.ch/research/topologies/`.

[21] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM.

[22] P. Jiang and G. Agrawal. Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 24:1–24:11, New York, NY, USA, 2017. ACM.

[23] P. Larson. Data Reduction by Partial Preaggregation. In *Proceedings of the 18th International Conference on Data Engineering, San Jose, CA, USA, February 26 - March 1, 2002*, pages 706–715, 2002.

[24] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDB J.*, 27(5):643–668, 2018.

[25] F. Liu, A. Salmasi, S. Blanas, and A. Sidiropoulos. Chasing Similarity: Distribution-aware Aggregation Scheduling (Extended Version). *CoRR*, abs/1810.00511, 2018.

[26] Y. Lu, A. Shanbhag, A. Jindal, and S. Madden. AdaptDB: Adaptive Partitioning for Distributed Joins. *PVLDB*, 10(5):589–600, 2017.

[27] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy. Parameter Hub: a Rack-Scale Parameter Server for Distributed Deep Neural Network Training. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*, pages

41–54, 2018.

[28] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, 2002.

[29] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 491–502, New York, NY, USA, 2003. ACM.

[30] S. Madden, R. Szewczyk, M. J. Franklin, and D. E. Culler. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *4th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2002), 20-21 June 2002, Callicoon, NY, USA*, pages 49–58, 2002.

[31] L. Mai, L. Rupprecht, A. Alim, P. Costa, M. Migliavacca, P. Pietzuch, and A. L. Wolf. NetAgg: Using Middleboxes for Application-specific On-path Aggregation in Data Centres. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 249–262, New York, NY, USA, 2014. ACM.

[32] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB*, 3(1):330–339, 2010.

[33] I. Müller, A. Arteaga, T. Hoefler, and G. Alonso. Reproducible Floating-Point Aggregation in RDBMSs. *CoRR*, abs/1802.09883, 2018.

[34] I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1123–1136, New York, NY, USA, 2015. ACM.

[35] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[36] O. Polychroniou, R. Sen, and K. A. Ross. Track Join: Distributed Joins with Minimal Network Traffic. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1483–1494, New York, NY, USA, 2014. ACM.

[37] P. Raghavendra and D. Steurer. Graph Expansion and the Unique Games Conjecture. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 755–764. ACM, 2010.

[38] P. Raghavendra, D. Steurer, and M. Tulsiani. Reductions Between Expansion Problems. In *Computational Complexity (CCC), 2012 IEEE 27th Annual Conference on*, pages 64–73. IEEE, 2012.

[39] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, 6(11):1080–1091, 2013.

[40] W. Rödiger, S. Idicula, A. Kemper, and T. Neumann. Flow-Join: Adaptive Skew Handling for Distributed Joins over High-speed Networks. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 1194–1205, 2016.

[41] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-Speed Query Processing over High-Speed Networks. *PVLDB*, 9(4):228–239, 2015.

[42] W. Rödiger, T. Mühlbauer, P. Unterbrunner, A. Reiser, A. Kemper, and T. Neumann. Locality-sensitive Operators for Parallel Main-memory Database Clusters. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 592–603, 2014.

[43] V. Satuluri and S. Parthasarathy. Bayesian Locality Sensitive Hashing for Fast Similarity Search. *PVLDB*, 5(5):430–441, 2012.

[44] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 25–36, 2003.

[45] A. Shatdal and J. F. Naughton. Adaptive Parallel Aggregation Algorithms. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 104–114, New York, NY, USA, 1995. ACM.

[46] E. Vermote-NASA GSFC and MODAPS SIPS - NASA. (2015). MOD09 MODIS/Terra L2 Surface Reflectance, 5-Min Swath 250m, 500m, and 1km. NASA LP DAAC.

[47] E. Vermote-NASA GSFC and MODAPS SIPS - NASA. (2015). MOD09A1 MODIS/Surface Reflectance 8-Day L3 Global 500m SIN Grid. NASA LP DAAC.

[48] L. Wang, M. Zhou, Z. Zhang, M. Shan, and A. Zhou. NUMA-Aware Scalable and Efficient In-Memory Aggregation on Large Domains. *IEEE Trans. Knowl. Data Eng.*, 27(4):1071–1084, 2015.

[49] J. L. Wolf, P. S. Yu, J. Turek, and D. M. Dias. A Parallel Hash Join Algorithm for Managing Data Skew. *IEEE Trans. Parallel Distrib. Syst.*, 4(12):1355–1371, Dec. 1993.

[50] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen. Handling Data Skew in Parallel Joins in Shared-nothing Systems. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1043–1052, New York, NY, USA, 2008. ACM.

[51] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 379–390, New York, NY, USA, 2004. ACM.

[52] Y. Ye, K. A. Ross, and N. Vesdapunt. Scalable

Aggregation on Multicore Processors. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, pages 1–9, New York, NY, USA, 2011. ACM.

[53] `https://www.yelp.com/dataset/documentation/json`.

[54] Y. Yu, P. K. Gunda, and M. Isard. Distributed Aggregation for Data-parallel Computing: Interfaces and Implementations. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 247–260, New York, NY, USA, 2009. ACM.

[55] E. Zamanian, C. Binnig, and A. Salama. Locality-aware Partitioning in Parallel Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 17–30, New York, NY, USA, 2015. ACM.