EF-dedup: Enabling Cooperative Deduplication at the Network Edge

Abstract—Deduplication is a popular data reduction technique for modern datacenters. Since the technique requires significant system resources, operators typically allocate dedicated appliances and network for its use, often in the same datacenter where the application is running. However, the recent developments of edge computing use-cases such as 5G and IoT services lead to new challenges, wherein, instead of just a few large-scale datacenters, we now have a hybrid cloud where hundreds, if not thousands, of small and resource-constrained datacenters, communicating with a few central cloud locations. Given their high costs, it is practically infeasible to place dedicated appliances in every edge site to enable local deduplication. At the same time, we need to be extra cautious regarding WAN bandwidth usage and hence cannot simply send raw data to the cloud for deduplication. Hence, cooperation among participating edge nodes to perform deduplication is desirable. However, the problem is very challenging since we need to jointly optimize required storage space and incurred network cost, while maintaining a competitive deduplication ratio. In this paper, we propose a novel distributed deduplication strategy, called EF-dedup, specifically tailored to the hybrid cloud environment. To minimize WAN bandwidth usage towards central cloud locations, we exploit the recent advance of modern distributed storage system to store hash indexes in an intelligent way, and send data to the central cloud locations only when it is really necessary. We formulate a problem that captures the unique trade-off between required storage space and network usage. Further, we propose an efficient heuristic to this NP-Hard problem that optimally partitions edge sites with similar data to enable edge deduplication. We apply our technique to an existing deduplication tool (Duperemove) with a popular distributed storage system (Cassandra). We deployed our prototype implementation to a real hybrid cloud, composed of one in a corporate lab and the other in AWS, to demonstrate the efficacy of our technique. Our experimental/simulation results validate our trade-offs and demonstrate that our technique can be 30–50% better than less involved heuristics.

I. INTRODUCTION

Data deduplication is a popular technique to optimize storage space in modern datacenters. Broadly speaking, deduplication is the process of splitting files into smaller data chunks and storing only unique chunks across the files. To detect duplicates, a typical deduplication system needs to maintain a hash index data structure in a certain form. The technique has been extensively studied in academia and, furthermore, has enjoyed its commercial success in large datacenters [?]. Many implementations of the technique require non-trivial system resources, e.g., cpu, memory, network and storage space. Therefore, operators often allocate dedicated hardware, e.g., in the form of appliances with a carefully engineered network for deduplication.

However, the recent development of edge computing mandates a rethinking of this practice. An edge network consists of hundreds, if not thousands, of small and resource-constrained edge servers/nodes (e.g., a half rack deployed in a central office location in a city) that are interconnected at the network edge. These edge nodes form edge clouds to provide users with very low latency communication, store and process data from various sources such as smart mobile devices, connected cars, sensors at home and/or drones, and communicate with more powerful central cloud datacenters for more involved processing or accumulating data. Large telecommunication companies are already building this edge cloud infrastructure [?], [?]. Current examples include IoT data management applications, virtualized RAN for 5G, augmented/virtual reality application, and some virtual network functions like consumer provider edge routers (CPE). The amount of data generated by these applications/devices is expected to grow in ZetaByte scale by 20xx [?].

The main rationale of edge computing is that computing should happen at the proximity of data sources. In this paper, we propose a new technique, collaborative Edge-Facilitated Deduplication (EF-Dedup), to push data deduplication to the network edge. To this end, our solution leverages several

1anonymous due to the review policy
978-1-4799-3360-0/14/$31.00 ©2014 IEEE

2In this paper, “server” and “node” are used interchangeably.
unique features of edge computing. (i) While it is impractical to have dedicated deduplication appliances in every edge node due to resource constraints, the ad hoc-like connected edges provide the opportunity for edge nodes to cooperate and deduplicate data in a distributed fashion. This allows us to leverage the network and computing power on “every thing” at the edge and achieve much higher deduplication throughput\(^3\) compared with solitary cloud approaches. (ii) Deduplicating close to data sources enables us to detect and eliminate duplicates in an earlier stage (i.e., at network edge), thus suppressing duplicated data transfer that would otherwise be transmitted across the WAN to central clouds. It can significantly reduce the usage of WAN bandwidth and release the burdens for expensive cloud uplink provisioning. (iii) Data flows generated by edge devices are often geographically correlated, e.g., sensors operating in the same environment and VR cameras located in close vicinity. The high degree of correlation provides the promise of space efficiency even if data flows are deduplicated only locally and distributively at network edge, without relying on cross-network processing.

To the best of our knowledge, this is the first systematic study of collaborative edge-centric data deduplication. It is easy to see that jointly processing all data flows arriving at the edge is practically impossible, due to prohibitive network traffic, replication/recovery, and difficulties to provide consistency guarantees in distributed edges. Therefore, to enable a distributed, yet collaborative effort, we partition edge nodes into disjoint clusters (which we refer to as Distributed-Deduplication-rings or D2-rings) and maintain a local (chunk) index table to deduplicate each cluster independently. Within each D2-ring, edge nodes collaborate to jointly deduplicate incoming data, and the storage of local index tables are also shared even by edge nodes in the D2-ring to avoid per-node space limitations and to facilitate parallel deduplication of concurrent data flows. While it is tempting to partition nodes with similar data flows into the same D2-rings to maximize the deduplication ratio and space efficiency, doing so may result in high network overhead and low throughput, due to intensive cross-node index table lookup if the nodes are located in distant edge clouds. Our approach is illustrated in Fig. ??.

Consider 5 edge nodes that are connected by 2 links with different network cost, and their data flows are comprised of sequences of data chunks and possess different level of similarity. Clearly, partitioning nodes \{1, 3, 5\} and \{2, 4\} together maximizes the deduplication with a total of 16 unique chunks from the two D2-rings. However it causes high network cost, in particular, between nodes 1 and 5. On the other hand, deduplicating each edge cloud separately achieves minimum network cost, yet it is not space efficient with 21 unique chunks. An optimal partitioning of edge nodes must take into account both network cost and data flow similarity. Leveraging this tradeoff is the pivotal point to unlock the potential of collaborative edge-centric deduplication.

The goal of this paper is minimize the network cost due to distributed index table storage and lookup, while maintaining a competitive deduplication ratio, through our proposed EF-Dedupe. It is a very challenging problem as we need to analyze the novel tradeoff between the space efficiency of deduplication and the overall network cost incurred by distributed, yet collaborative deduplication, which becomes more salient in practice when we cross the boundaries of existing edge clouds (Sec. V). The collaborative EF-Dedupe is specifically tailored to the edge computing paradigm by addressing the following problems: First, how to leverage the tradeoff to develop novel collaborative deduplication solutions that are both edge-centric and optimal? Second, what system techniques will enable distributed deduplication across resource constrained edge nodes, often connected by unstable networks?

To address the first challenge, we formulate an optimization problem to partition the edge nodes into collaborative D2-rings, considering: (i) the data similarities across the nodes (i.e., deduplication ratio will improve as more data flows belonging to the same are similar), and (ii) the network cost in performing deduplication across edge nodes (i.e., the deduplication throughput will decrease when performing deduplication across edge nodes that are distant from each other). To this end, we capture the data similarity across different edge nodes through a novel, hierarchical data model. In particular, we first construct a set of independent chunk pools, which intuitively represent commonly occurring data sources such as chunk pools typical of Windows, Linux, and dictionaries. Given these chunk pools, the data flow generated by each node can be statistically constructed by randomly drawing data from the chunk pools, according to a pre-determined (empirical) probability distribution function. The proposed hierarchical data model is validated using real-world datasets with an average error less than 4% (Sec. III-A). Based on the model, we formulate the joint space and network optimization for edge-centric deduplication, which is proven to be NP-hard through transformation of minimum \(k\)-cut problem. We develop efficient heuristics to the optimization (for both balanced and unbalanced partitioning problems). We also prove that the proposed heuristics is optimal when the number of chunks pools is equal to two, and in general, achieves an approximate ratio \(1 + \left(\frac{S}{Z} - 1\right) \left(1 - \frac{1}{2}\right)\), where \(S\) is the size of all chunk pools, \(U^*\) the optimal storage space, and \(Z\) the size of D2-rings. Our models and analysis pave the way for a systematic framework on collaborative, edge-centric deduplication.

While our algorithms optimize edge-node partitions with provable performance guarantees, another major challenge

\(^3\)Here, the deduplication throughput is the amount of data deduplicated data within a certain timeframe experienced by the client(s).
lies in designing a system to perform distributed deduplication within each resulting partition (i.e., D2-ring), which could often contain nodes across different edge clouds. In EF-dedup we address this challenge through a novel design intuition: given the resource constraints of individual edge node in each D2-ring and the need for parallel data flow processing, why not maintain the index table consisting of chunk hashes in a distributed storage system (such as Cassandra [?]) spread across the associated edge nodes? The distributed storage system provides a way to harness resources (including cpu, memory and storage space) that already exist on edge nodes, in an effective fashion with enormous flexibility and scalability. This empowers EF-dedup with the ability to leverage various resource from "things" at the edge of the network. In this paper, we realize this intuition by modifying an existing deduplication tool, duperemove [?] to use Cassandra for its index structure. Our experiments on two real-world clouds, one in a corporate lab environment and the other in AWS, with many real-world datasets such as VM images and dictionaries shows that EF-dedup provides 30–50% better performance than relatively simpler strategies while maintaining very high deduplication ratio. In summary, we make the following contributions:

- To push deduplication to network edge, we identify a novel tradeoff in the problem space and formulate an optimization problem to partition edge nodes considering both space efficiency and network cost (Sec. II). The formulation is enabled by a new hierarchical model for correlated data flows.
- We prove that the formulated problem is NP-Hard by transformation from minimum k-cut and provide efficient heuristics to the problem. Approximate ratio of the proposed heuristics are obtained in closed-form (Sec. III-B).
- We present an end-to-end system design to realize EF-dedup wherein we maintain the index structure of each deduplication D2-ring in a distributed storage system deployed across multiple edge clouds. We implemented EF-dedup by modifying duperemove [?] to use Cassandra for its index structure (Sec. IV).
- We validate our system model and algorithms through extensive simulations and experiments performed on two real clouds using real-world datasets (Sec. V). Significant improvement on system throughput is achieved yet with high deduplication ratio.

II. SYSTEM MODEL AND PROBLEM FORMULATION

We consider a distributed edge network consisting of $N$ edge nodes, e.g., VMs in cloudlets, fog/edge equipments, and mobile devices, denoted by $N' = \{1, 2, \ldots, N\}$. The edge nodes generate data flows, such as VM/system backup, smartphone images and sensing data, which need to be stored in the cloud.

In this paper, we consider a novel approach to enable edge-centric deduplication, which eliminate duplicate copies of the incoming data flows at the network edge, so that the amount of data that must be stored or transferred to the cloud can be greatly reduced, given that the same byte pattern (i.e., data chunks) may occur frequently in different data flows. In particular, we partition the edge nodes into disjoint Distributed-Deduplication rings (denoted as D2-rings), that may traverse different edge clouds, based on their network conditions and data similarity. Each D2-ring independently performs the deduplication process, where unique chunks generated from data flows associated with the D2-ring are identified and transferred to the central cloud for storage. As the process continues, hash values of the unique chunks are stored locally and distributely on all edge nodes associated with the D2-ring, so that any incoming chunks are compared to the hash values to determine if a redundant chunk has occurred.

To jointly minimize the storage space and network cost, we consider each edge node $i$ as a data source that generates data flows at a rate of $R_i$ chunks per second. Note that, on the D2-rings (at the edge), we are only concerned with the storage of the chunk hashes to enable deduplication, since unique data chunks are sent to and stored on the cloud. To model the spatial and temporal correlation both within each data source and between different data sources, we assume that each chunk generated by source $i$ is randomly drawn from $K$ disjoint chunk pools, which is denoted by $C_1, C_2, \ldots, C_K$, with known probabilities $p_{i1}, p_{i2}, \ldots, p_{iK}$. For example, $C_1$ represents chunks typical for Windows OS, $C_2$ for Linux, and $C_3$ for chunks shared by the two systems due to common applications. We further assume that each chunk of source $i$ is independently generated by randomly selecting a chunk pool with probabilities $\{p_{ik}, \forall k\}$ and then choosing a chunk within the selected pool with a uniform distribution. We denote the probability vector $P_i = [p_{i1}, p_{i2}, \ldots, p_{iK}]$ as the characteristic vector of source $i$, which quantifies the statistics of its data flow.

In our system model, data generated by correlated sources have the same probabilities of selecting chunks from the $K$ chunk pools, resulting in higher level of redundancy. These probabilities can be empirically obtained using data source profiling and/or estimated through meta data (Sec. III-A). If a set of data sources (i.e., edge nodes) are clustered into a single D2-ring, which is denoted by $\mathcal{P}$ where $\mathcal{P} \subseteq N'$, their data flows are jointly deduplicated. Let $\Omega(\mathcal{P})$ be the expected overall deduplication ratio of all sources in $\mathcal{P}$, i.e., the original data size divided by the deduplicated storage size. The expected storage space required for D2-ring $\mathcal{P}$ during an interval of $T$ seconds is given by

$$U(\mathcal{P}) = \frac{1}{\Omega(\mathcal{P})} \sum_{i \in \mathcal{P}} R_i T$$  \hspace{1cm} (1)
where \( R_i \) is the data rate of source \( i \).

While incorporating more edge nodes into a single D2-ring improves space efficiency due to increased chance of finding redundant chunks, it also incurs higher network cost during the deduplication process, because the hash values of identified unique chunks are distributedly stored on peer edge nodes in the D2-ring. As its size increases, a higher fraction of chunk hash values are stored on non-local edge nodes, resulting in higher network cost for hash lookups when new data chunks arrive. Let \( \gamma \) be the (chunk) replication factor in the D2-ring, i.e., each unique chunk is stored on \( \gamma \) distinct edge nodes. We consider a D2-ring \( P \) that has size \( |P| \). When chunks are uniformly distributed on edge nodes associated with the D2-ring \( P \) (e.g., using a distributed hash table), the probability of requiring a non-local hash lookup for any incoming data chunk is \( 1 - \gamma/|P| \). Let \( v_{ij} \) be the network cost of a non-local hash lookup forwarded from node \( i \) to node \( j \); e.g., it can be measured by the necessary bandwidth or network delay of the non-local hash lookup. The total network cost during the deduplication process of D2-ring \( P \) in an interval of \( T \) seconds is thus

\[
V(P) = \sum_{i \in P} \sum_{j \neq i \in P} v_{ij} \frac{R_i T (1 - \gamma/|P|)}{|P| - 1},
\]

where each non-local hash lookup has equal probability \( 1/(|P| - 1) \) to be processed by peer edge nodes \( \{ j : j \neq i, j \in P \} \) in the D2-ring.

Our goal is to find a partition of all edge nodes \( N \) into \( M \) disjoint D2-rings, i.e., \( P_1, P_2, \ldots, P_M \) satisfying \( \cup_s P_s = N \), to minimize jointly the total required storage space \( \sum_s U(P_s) \) and network cost \( \sum_s V(P_s) \). Let \( \alpha \) be a tradeoff factor quantifying the relative importance of network cost to storage space, i.e., each unit network cost is equivalent to \( \alpha \) units of storage space increment. We formulate the joint Storage and Network Optimization in Distributed Deduplication (SNOD2) as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_s U(P_s) + \alpha \sum_s V(P_s) \\
\text{s.t.} & \quad \frac{1}{\Omega(P_s)} \cdot \sum_{i \in P_s} R_i T, \\
& \quad \sum_{i \in P_s} \sum_{j \neq i \in P_s} v_{ij} \frac{R_i T (1 - \gamma/|P_s|)}{|P_s| - 1} \\
\text{var.} & \quad P_s, \ \forall s,
\end{align*}
\]

where \( P_s, \ \forall s \) forms a disjoint partition of the edge nodes.

## III. Algorithms

In this section, we first quantify the storage space efficiency (i.e., deduplication ratio function \( \Omega(P_s) \)) for a given partition \( P_1, P_2, \ldots, P_M \). Then, we present a novel technique to estimate the characteristic function of a given set of data sources, that is required to solve the SNOD2 problem. This technique has significant practical implications even outside the realm of this paper, since it provides a simple way to estimate the chunk distribution functions of any set of data sources by sampling just a few files (Section V-D).

Finally, we show that the SNOD2 problem under our system model is NP hard and propose a greedy algorithm to solve a balanced SNOD2 where all partitions (i.e., D2-rings) have equal size. The algorithm can be proven optimal when the number of disjoint chunk pools \( K = 2 \), and has a guaranteed competitive ratio when \( K > 2 \). Then, we develop two algorithms for arbitrary SNOD2 by leveraging greedy and hierarchical matching heuristics.

### A. Estimating Source Characteristic Functions

To solve the SNOD2 problem, we first need to quantify the deduplication ratio of any given partition based on the chunk pools that best represent the sources and their characteristic functions.

**Theorem 1:** We assume that each chunk generated by data source is randomly drawn with known probabilities \( p_{i1}, p_{i2}, \ldots, p_{iK} \) from \( K \) disjoint chunk pools \( C_1, C_2, \ldots, C_K \) where \( s_k \) is the size of chunk pool \( C_k \). For data sources with characteristic vectors \( \{P_s, \ \forall i\} \), the deduplication ratio of D2-ring \( P_s \) is given by

\[
\Omega(P_s) = \frac{\sum_{i \in P_s} R_i T}{\sum_{k=1}^K s_k (1 - \prod_{i \in P_s} g_{ik})},
\]

where \( g_{ik} = (1 - p_{ik}/s_k) R_i T \).

**Proof:** It is easy to see that the original data flow size is \( \sum_{i \in P_s} R_i T \) for an interval of \( T \) seconds. Without loss of generality, we consider a data chunk in pool \( C_k \). Based on our data flow construction model in Section II, this chunk is selected when source \( i \) generates a new chunk, with probability \( p_{ik}/s_k \). The probability that the chunk is never selected by source \( i \) during an interval \( T \) is given by

\[
g_{ik} = (1 - p_{ik}/s_k) R_i T,
\]

Since all chunks in pool \( C_k \) are selected with the same probability, the expected number of distinct chunks drawn from \( C_k \) by all sources is thus \( s_k (1 - \prod_{i \in P_s} g_{ik}) \), whose summation over all chunk pools \( C_1, C_2, \ldots, C_K \) yield the total required storage space after deduplication.

The key question here, however, is: **For an unknown set of sources, how do we know the chunk pools that best represent them and the characteristic functions of the sources?** In algorithm 1, we address this problem by first sampling a few files at random from each source and exhaustively searching across all possible values of the main parameters that we need to obtain, viz. the number of chunk pools, the size of each chunk pool and the chunk distribution or characteristic function of each source.
their efficacy is an exciting avenue for future research, the real ones.

the simulated deduplication ratios are 4 combo combination and the model-calculated values, is 4 MSE, between real deduplication ratio averages for each combination are the closest to the real ones. The parameters, so that the model-calculated deduplication ratios do enumeration search to find the most appropriate model A2A4B6B8, A0B2B4B10, A2A6B0B8 and so on. Then we from source 2, we have combination enumerations such as combination including 2 updates from source 1 and 4 updates of the enumeration combinations options. For example, for form combinations, and record deduplication ratio of each combination. For any combination, we take average of all from source 2, we have combination enumerations such as A2A4B6B8, A0B2B4B10, A2A6B0B8 and so on. Then we do enumeration search to find the most appropriate model parameters, so that the model-calculated deduplication ratios for each combination are the closest to the real ones. The MSE, between real deduplication ratio averages for each combination and the model-calculated values, is 4%. It means the simulated deduplication ratios are 4% more or less than the real ones.

Designing better heuristics for this estimation and proving their efficacy is an exciting avenue for future research, especially because, estimating the chunk distributions of data sources can be useful across a wide range of applications. For example, it can help guide how the chunk sizes should be selected for deduplication (e.g., to pick the chunk size that minimizes estimation error) or guide what should be maintained in the deduplication cache (e.g., to maintain the chunks that appear with higher probability in the chunk pools).

In the following section, given these estimated parameters, we describe how we provide efficient heuristics for the SNO2 problem.

B. Our Proposed Solution to SNO2

To show that the SNO2 problem is NP-Hard, we first apply Theorem 1 and rewrite the SNO2 problem as follows:

\[
\text{minimize } \sum_s U(P_s) + \alpha \sum_s V(P_s) \tag{8}
\]

s.t. \[
U(P_s) = \sum_{k=1}^{K} s_k \left(1 - \prod_{i \in P_s} g_{ik}\right), \tag{9}
\]

\[
g_{ik} = (1 - p_{ik}/s_k)^{R_{i}T}, \quad \forall i, k \tag{10}
\]

\[
V(P_s) = \sum_{i:j \in P_s, j \neq i} \sum_{i,j \in P} v_{ij} R_{i}T \left(1 - \gamma/|P_s|\right) / |P_s|-1 \tag{11}
\]

\[\text{var. } P_s, \forall s.\]

**Theorem 2:** The proposed SNO2 problem is NP hard.

**Proof:** We show that the minimum k-cut problem (which is known to be NP hard when k is an input variable \(k\)) can be transformed into a SNO2 problem with zero network cost.

Consider an undirected graph \(G = (V, E)\) with an assignment of weights to the edges, denoted by \(w(v_1, v_2)\) for any edge \((v_1, v_2) \in E\). The minimum k-cut problem partitions vertices in \(V\) into \(k\) disjoint sets, \(P_1, P_2, \ldots, P_k\), while
minimizing the sum of removed edge weights:
\[
\sum_{(v_1, v_2) \in E} w(v_1, v_2) \cdot 1_{\{\exists P_s : v_1 \in P_s, v_2 \in P_s\}},
\]
where \(1_{\{\exists P_s : v_1 \in P_s, v_2 \in P_s\}}\) is an indicator function that is equal to 1 if vertices \(v_1, v_2\) are in the same partition, and 0 otherwise.

We construct a SNOD2 problem where each vertex in \(V\) is considered as a data source and each edge in \(E\) corresponds to a chunk pool. More precisely, for an edge \((v_1, v_2) \in E\), we construct chunk pool \(k = v_1N + v_2\), which has size \(s_k = w(v_1, v_2)/(1-c)^2\), for some constant \(c \in (0, 1)\). Let \(d(v_1)\) be the degree of a vertex \(v_1 \in V\). We construct a characteristic vector for data source \(v_1\) by setting \(p_{v_1, k} = 1/d(v_1)\) if the edge corresponding to chunk pool \(k\) contains vertex \(v_1\) (or more formally, if there exists an edge \((v_1, v_2) \in E\) satisfying \(k = v_1N + v_2\), and \(p_{v_1, k} = 0\) otherwise. For each source \(v_1\), we choose a data rate \(R_{v_1} = \log(c)/[T \cdot \log(1-p_{v_1}/s_k)]\) for some positive \(T\) and the same constant \(c\). Finally, all network costs are assumed to be zero.

Now we prove that the constructed SNOD2 problem finds a disjoint partition of \(V\) to minimize the same objective function in (12). Let \(N = |V|\) be the number of sources/vertices. Using (8) and (9), the optimization objective of the SNOD2 problem becomes
\[
\sum_{s} \sum_{k} s_k \left(1 - \prod_{i \in P_s} g_{ik}\right) = \sum_{k} s_k \sum_{s} \left(1 - \prod_{i \in P_s} g_{ik}\right) = \sum_{k} s_k \left(N - \sum_{s} \prod_{i \in P_s} g_{ik}\right)
\]
which can be consolidated using the indicator function
\[
\sum_{s} \prod_{i \in P_s} g_{ik} = \sum_{i \neq v_1, v_2} \prod_{i \neq v_1, v_2} g_{ik} + g_{v_1, k} + g_{v_2, k} = N - 2 + 2c, \text{ if } \exists P_s : v_1 \in P_s, v_2 \in P_s,
\]
\[
\sum_{s} \prod_{i \in P_s} g_{ik} = N - 1 + c^2, \text{ otherwise},
\]
\[
\sum_{s} \prod_{i \in P_s} g_{ik} = N - 1 + c^2, \text{ otherwise},
\]
\[
\sum_{s} \prod_{i \in P_s} g_{ik} = N - 1 + c^2, \text{ otherwise},
\]
\[
\sum_{s} \prod_{i \in P_s} g_{ik} = N - 1 + c^2, \text{ otherwise},
\]
plugging this into the last step of (15), we have
\[
\sum_{s} \sum_{k} s_k \left(1 - \prod_{i \in P_s} g_{ik}\right) = \sum_{k} s_k(1 - c^2) + \sum_{k} s_k(1 - c)^2 \cdot 1_{\{\exists P_s : v_1 \in P_s, v_2 \in P_s\}} = \sum_{k} s_k(1 - c^2) + \sum_{k} w(v_1, v_2) \cdot 1_{\{\exists P_s : v_1 \in P_s, v_2 \in P_s\}}
\]
where we used \(s_k = w(v_1, v_2)/(1-c)^2\) in the last step, and \(v_1, v_2\) are the two vertices belonging the edge corresponding to chunk pool \(k = v_1N + v_2\). Notice that \(\sum_{k} s_k(1 - c^2)\) is a constant not affected by the partitioning, and that the summation over index \(k\) is the same as \(v_1, v_2\) (due to one-to-one correspondence between chunk pools and edges in our construction). We conclude that any solution to the SNOD2 problem solves the minimum \(k\)-cut problem, which implies that SNOD2 is also NP hard.

1) Our Greedy Algorithm for Balanced Partitioning: We consider a SNOD2 problem with balanced partitioning. More precisely, for a given set of \(N = Z \cdot M\) edge nodes, we partition them into \(M\) disjoint D2-rings, each containing an equal number of edge nodes \(|P_s| = Z\) \(\forall s\). Using the same proof in Theorem 1, it is easy to show that the balanced SNOD2 problem is also NP hard, since balanced \(k\)-cut problem is known to be NP hard. We develop a greedy algorithm for the problem and analyze its competitive ratio.

In the proposed algorithm for balanced SNOD2 problem, we iteratively select \(Z\) edge nodes \(P\) that has the smallest aggregate cost \(U(P) + \alpha \cdot V(P)\) among all remaining nodes, and group them into a new D2-ring. The process continues until there is no edge node left and all \(M\) D2-rings are created. It is easy to see that this algorithm requires computing the aggregate cost of \(N\)-choose-\(Z\) possible partitions satisfying \(\forall P \in \mathcal{P} |P| = Z\) \(\forall s\). In the algorithm summarized in Algorithm 4, we sort all \(N\)-choose-\(Z\) partitions in an ascending order with respect to their aggregate costs. Then, in each iteration, we choose a remaining available partition with the smallest aggregate cost. Once a partition \(P\) is selected, all other partitions that have overlap with \(P\) are eliminated. The algorithm solves the balanced SNOD2 problem in exactly \(M\) iterations, with a computation complexity \(O(N\text{-choose-}Z) \cdot \log \lfloor N\text{-choose-}Z \rfloor\) due to sorting the possible partitions.

Theorem 3: For equal network costs, the proposed algorithm for balanced SNOD2 problem is optimal when \(K = 2\), and its required storage space achieves a competitive ratio \(1 + (\frac{\sqrt{5} - 1}{2}) (1 - \frac{1}{Z})\) when \(K > 2\), where \(U^*\) is the optimal

4If the number of edge nodes \(N\) is not an integer multiple of \(M\), we can add dummy nodes with zero data rate to satisfy this condition.
Algorithm 2 Greedy Balanced Partitioning Algorithm

foreach cluster $i$

    foreach Z-node partition $P_j$ in remaining nodes
        Calculate the aggregate cost $U(P_j) + \alpha \cdot V(P_j)$;
        Find $P_{min}$ with the smallest aggregate cost;
    end

    $P_i = P_{min}$;
    Remove Z nodes of partition $P_i$ from remaining nodes;
end

storage space and $S = \sum_k s_k$ is the total size of all chunk pools.

Proof: When network costs are equal, the SNOD problem reduces to a minimization of required storage space $\sum_s \sum_k s_k (1 - \prod_{i \in P} g_{ik})$ over a balanced partitions $P_1, \ldots, P_M$. Since $\sum_s \sum_k s_k$ is a constant, the problem boils down to maximizing

$$
\sum_s \sum_k s_k \prod_{i \in P_s} g_{ik} = \sum_s \sum_{i \in P_s} g_{ik} \left( \sum_{i \in P_s} \prod_{j \in P_s} g_{ij} \right).
$$

(17)

Proof of optimality when $K = 2$. Consider the problem of partitioning $N = MZ$ positive numbers $g_{11}, g_{22}, \ldots, g_{NN}$ into $M$ equal-size subsets $P_1, P_2, \ldots, P_M$, to maximize the sum of products, $\sum_s \sum_k g_{ik}$, which is the last term in (17). It can be shown that the optimal solution to this problem is to sort the numbers in descending (or ascending) order (denoted by $g_{1k}^*, g_{2k}^*, \ldots, g_{Nk}^*$) and then group every $M$ adjacent numbers into each partition, i.e., $P_s = \{g_{s_i-N+1,k}^*, g_{s_i-N+2,k}^*, \ldots, g_{s_i,N,k}^*\}$ for $s = 1, \ldots, M$.

We prove this by constructing a contradiction. Assume that $\sum_s \sum_k g_{ik}$ is maximized by some optimal partition, in which there exist $x_1, y_1 \in P_1$ and $x_2, y_2 \in P_2$, but $x_1 > y_1$ and $x_2 < y_2$ (i.e., the numbers $x_1, y_1, x_2, y_2$ are not sorted). Without loss of generality, we assume $\prod_{i \in P_1} (x_1, y_1) g_{ik} > \prod_{i \in P_2} (x_2, y_2) g_{ik}$. Therefore, we swap $y_1 \in P_1$ and $y_2 \in P_2$, it yields a strictly higher objective value, since $x_1y_2 \prod_{i \in P_1} (x_1, y_1) g_{ik} + x_2y_1 \prod_{i \in P_2} (x_2, y_2) g_{ik} > x_1y_2 \prod_{i \in P_1} (x_1, y_1) g_{ik} + x_2y_2 \prod_{i \in P_2} (x_2, y_2) g_{ik}$. This contradicts with the optimality of partitions $P_1, P_2$ in our assumption.

Notice that when $K = 2$, we have $g_{11} + g_{22} = 1 \forall i$. Therefore, our proofed greedy algorithm that partitions the edge nodes by sorting $g_{11}, g_{22}, \ldots, g_{N1}$ (for $k = 1$) in a descending order automatically sorts $g_{22} = 1 - g_{11}, \ldots, g_{N2} = 1 - g_{N1}$ in an ascending order. The solution simultaneously optimizes $\sum_s \sum_k g_{ik}$ for both $k = 1$ and $k = 2$, which is an optimal solution maximizing (17) and thus the balanced SNOD2 problem.

Proof of competitive ratio when $K > 2$. Let $\phi(P_s) = \sum_k s_k \prod_{i \in P_s} g_{ik} \forall s$. We consider the maximization of (17), which is $\sum_s \phi(P_s)$. In the following, We use $P_1, \ldots, P_M$ to denote the feasible partition obtained by our proposed greedy algorithm, and $P_1^*, \ldots, P_M^*$ for the optimal partition. We first show that

$$
\sum_s \phi(P_s) \geq \frac{1}{Z} \sum_s \phi(P_s^*),
$$

(18)

where $Z$ is the size of each partition. We denote the optimal objective value above by $\Gamma(N) = \sum_s \phi(P_s^*)$ for $N = \cup_s P_s^*$. We prove the result by induction. Consider $M \leq Z$. Our greedy algorithm first finds a group of $Z$ sources, $P_1$, which has the largest sum of product, i.e., $\phi(P_1) \geq \phi(P)$ for any $\phi(P) \in \mathcal{N}$. It is easy to see that

$$
\sum_s \phi(P_s) \geq \phi(P_1) \geq \sum_s \phi(P_s^*) / M \geq \sum_s \phi(P_s^*) / Z,
$$

(19)

where we use the facts that $P_1$ has the largest sum of product and that $M \leq Z$. Now suppose that the result holds for all $M \leq M_0$. We consider $M = M_0 + 1$. The Z nodes belonging to $P_1$ would be distributed in $x \leq Z$ distinct partitions in the optimal solution, i.e., $P_1^*, \ldots, P_x^*$ without loss of generality. It is easy to see that $P_2^*, \ldots, P_x^*$ offers an optimal partitioning for a subset of nodes $\cup_{s=1}^x P_s^*$. Similarly, $P_{x+1}^*, \ldots, P_M^*$ is optimal for a subset of nodes $\cup_{s=x+1}^M P_s^*$. Then, we have

$$
\sum_{s=1}^M \phi(P_s) = \phi(P_1) + \sum_{s=2}^M \phi(P_s)
\geq \frac{1}{Z} \sum_{s=1}^x \phi(P_s^*) + \frac{1}{Z} \Gamma(\cup_{s=2}^M P_s^*)
\geq \frac{1}{Z} \sum_{s=1}^x \phi(P_s^*) + \frac{1}{Z} \sum_{s=x+1}^M \phi(P_s^*)
\geq \frac{1}{Z} \sum_{s=1}^M \phi(P_s^*).
$$

(20)

The second step follows from the fact that $\phi(P_1) \geq \phi(P_s^*)$ for any $s$ due to our greedy algorithm, and from the induction assumption that $\sum_{s=2}^M \phi(P_s) \geq \Gamma(\cup_{s=2}^M P_s^*) / Z$ for any set of $(M - 1)Z = M_0Z$ nodes, i.e., $\cup_{s=2}^M P_s$, $\Gamma(N)$ denotes the maximum objective value by optimally partitioning nodes in $\mathcal{N}$. The third step holds because $\cup_{s=x+1}^M P_s^* \subset \cup_{s=2}^M P_s$, and thus a higher maximum objective $\Gamma$ value is always achieved when more nodes are added to the system. The fourth step uses the fact that the partition $P_{x+1}^*, \ldots, P_M^*$ is optimal for a subset of nodes $\cup_{s=x+1}^M P_s^*$. Finally, the last step uses $x \leq Z$. Therefore, the induction assumption holds for $M = M_0 + 1$.

Using $S = \sum_k s_k$ and the optimal storage space $U^*$, we
obtain the competitive ratio:
\[
S - (S - U^*) = 1 + \left( \frac{S}{U^*} - 1 \right) \left( 1 - \frac{1}{Z} \right).
\] (21)

This completes the proof of our theorem.

Theorem 3 implies that for \( K > 2 \), the competitive ratio of our greedy algorithm approaches 1 as \( S/U^* \) decreases, i.e., in a heavy-traffic system with more data sources and chunks.

2) Our Algorithms for Arbitrary Partitioning: We develop two heuristic algorithms for the SNOD2 problem with D2-ring/partition size. The first algorithm is based on a greedy approach that iteratively selects a remaining edge node with the smallest cost increment and places it into one of \( M \) existing D2-rings, while the second algorithm performs a sequence of minimum-weight matching between remaining edge nodes and D2-rings until there is exactly \( M \) groups left.

The greedy algorithm is summarized in algorithm 5. It begins with \( M \) empty D2-rings, \( P_1, P_2, \ldots, P_M \), and iteratively places a remaining node \( v \in V \) into ring \( P_s \), if
\[
\{v, s\} = \arg \min_{v \in V, s} \left[ U(P_s \cup \{v\}) + \alpha V(P_s \cup \{v\}) - U(P_s) - \alpha V(P_s) \right].
\] (22)

In order words, placing node \( v \) in D2-ring \( P_s \) results in the minimum cost increment. Then, we update \( V / \{v\} \rightarrow V \) and \( P_s \cup \{v\} \rightarrow P_s \). The process continues until all nodes are placed. It is easy to see that the greedy algorithm has a computation complexity \( o(N^2 \cdot M) \).

The hierarchical matching algorithm is summarized in [Add a pseudo code here]. Starting with \( N \) nodes, we perform a minimum-weight matching where the weight between two nodes \( v_1, v_2 \) is defined as the aggregate cost \( U(\{v_1, v_2\}) + \alpha V(\{v_1, v_2\}) \) (a dummy node can be added if \( N \) is an odd number). If we preserve the first \( \theta N \) matches that have the lowest weights, this step produces \( N - L \) partitions, each containing 1 or 2 nodes. Next, the same minimum-weight matching is carried out on the remaining partitions (with the weight between \( P_1, P_2 \) defined again as the aggregate cost \( U(\{P_1 \cup P_2\}) + \alpha U(\{P_1 \cup P_2\}) \)), and the first \( \theta \) matches with minimum weights are preserved. This procedure reduces the number of partitions by a factor of \( \theta \) in each step. The proposed algorithm converges in \( o(\log_{1-\theta}(N/M)) \) rounds.

IV. DESIGN AND IMPLEMENTATION

In this section, we present the system design for EF-dedup, in which we explore the novel intuition of maintaining chunk hashes/deduplication index across edge nodes (that may belong to different edge clouds) in a distributed key-value store (Cassandra [2]). We then present the implementation details of our system wherein we modify duperemove [2], one of the few open source deduplication tools that are commonly used, to incorporate the Cassandra hashing.

Algorithm 3 Arbitrary Partitioning Algorithm

```plaintext
foreach node i
  foreach cluster j
    Calculate cluster aggregate cost \( U(P_j \cup \{v\}) + \alpha V(P_j \cup \{v\}) \)
    Find the cluster \( P_{min} \) with the smallest cluster aggregate cost
    \( P_{min} = P_{min} \cup i \)
    Remove node \( i \) from remaining nodes
end
```

A. System Design and Implementation

In Figure 3 we depict our system architecture, with an example of five edge nodes E1 ~ E5 that are clustered into two independent D2-rings. Our system creates a Cassandra cluster (or ring) for each D2-ring to maintain the hash values for the data chunks. Cassandra, one of the most widely used distributed key-value stores, has important features that we need for implementing the D2-ring. First, a single logical database is spread across a cluster of edge nodes that have constrained storage resources. Second, the consistent hashing minimizes the key movements when edge nodes join or leave the cluster. Third, Cassandra supports data replication that
can be leveraged to maintain multiple copies of popular chunk hashes.

Each edge node runs our Dedup Agent, which performs the task of deduplicating the input files by maintaining the chunk hashes in the D2-ring’s Cassandra cluster. Each Cassandra ring is capable of maintaining multiple copies of chunk hashes depending on its replication factor. In Fig. 3(a), for example, data chunk A’s hash (= a) is stored in two nodes (i.e., E2 and E3) and data chunk B’s hash (= b) is also stored in two nodes (i.e., E1 and E2) because their Cassandra ring’s replication factor is two. Maintaining more copies of the hashes in Cassandra enables the Dedup agent to perform a local node-level lookup for hashes, but this also increases the amount of storage needed for the hashes.

Figure 3(b) shows how our Dedup Agent deduplicates files generated from diverse OSs such as Windows, Linux, and so forth. After splitting files into smaller chunks, the Dedup Agent computes the hash value of each chunk. The Dedup Agent then performs a lookup to determine if this hash value is present in the Cassandra cluster and only if it is not present it adds this new hash to the cluster and sends the data chunk corresponding to this hash to the central cloud. For example, in Figure 3(b), only nonpresent unique chunks (i.e., C, E) across the files are sent to the central cloud.

To implement our Dedup Agent, we modified duperemove, which is a user-space application program for finding duplicate extents (contiguous storage area in a file system). In its native form, duperemove only finds duplicate extents when the input files are located in a Btrfs file system. To enable our Dedup agent to operate on any file system, we modified duperemove (approximately 1200 lines of additional code) to retain only the functions related to hashing and chunking from the original code. Additionally, we replaced its Sqlite database with Cassandra, for reasons mentioned above. For connecting duperemove to Cassandra, we use the DataStax C/C++ driver.

V. EVALUATION

A. Experimental Setup

Our problem space has many dimensions to explore, such as the scale of network, delay, similarity of sources. To properly evaluate the performance of our distributed algorithm, we conducted extensive simulations and experiments in a real distributed environment.

The distributed environment includes DDAs(Distributed Dedup Agent) and CDAs(Cloud Dedup Agent), where DDAs are in Openstack-based clusters and CDAs are in Amazon EC2-based clusters. We create 20 VMs in Openstack to represent DDAs, and 4 VMs in Amazon EC2 to represent CDAs. Each VM has 4 VCPUs, 8 GB RAM and 20 GB virtual disk drive. The average data rate among DDAs is 1.7255 Gbps with average latency as 0.85ms, while the average data rate between DDAs and CDAs is 0.3774 Gbps with average latency as 12.2 ms. We use netem to add delay in traffic among clusters. Agents within the same cluster will not be added any delay. All results presented in this paper in each setup are average values of at least 20 runs.

B. Comparing EF-dedup and Other Algorithms

To demonstrate the benefits of the proposed EF-dedup strategy, we evaluate its performance under two metrics - deduplication ratio and system throughput - and compares it with two baseline strategies: Cloud-only and Cloud-assisted deduplication, which are illustrated in Fig. 1. We run parallel deduplication tasks on all 20 DDAs. Dedup ratio is defined as the original data size divided by the deduplicated size, while dedup throughput is measured by the amount of data processed per second by each DDA node.

We use two different data sets for our evaluation, including Linux Kernel Archives and synthetic images created by Vdbench, a command line utility for generating disk I/O workload. Specifically, Linux Kernel Archives have 4 large versions, and tens of subversions in each large version. We consider each update folder of a subversion as an input file for deduplication, which has size 700-800MB. This Linux Kernel data set is used for evaluations in Fig. 4(a) and Fig. 4(b). On the other hand, Vdbench is a command line utility specifically created to generate disk I/O workloads to be used for validating storage performance and storage data integrity. We use Vdbench to create files with desired deduplication ratio and assign them to different DDAs. For EF-dedup, all DDAs are evenly partitioned into two D2-rings. For both Cloud-only and Cloud-assisted, 4 CDSs are set up on the cloud and they evenly share the workload sent from edge DDAs.

Fig. 4(a) shows that EF-dedup can significantly improve the system throughput over cloud-only and cloud-assist strategies, as the network size increases from 4 to 20 DDAs. Cloud-only strategy consistently has low throughput since all raw data has to be forwarded to the cloud for centralized deduplication, thus encountering a performance bottleneck due to constrained upload bandwidth. While Cloud-assisted strategy pushes deduplication to the edge DDAs, its throughput is still limited by the need of frequent message passing between edge DDAs and cloud CDAs for chunk/hash lookup. The proposed EF-dedup outperforms cloud-only and cloud-assist by up to 190.14% and 33.32%, respectively. This is because EF-dedup enables distributed deduplication within each D2-ring, requiring only local message exchange on the edge, between DDAs in the same D2-ring. Thus, its throughput grows almost linearly as the network size increases, demonstrating EF-dedup’s unique ability to fully leverage the distributed nature of edge network, for rapid deduplication.
(a) For varying network size (i.e., the number of DDAs), EF-dedup significantly improves throughput over Cloud-only and Cloud-assisted.

(b) As the latency between cloud and DDAs on the network edge increases, EF-dedup continues to outperform both Cloud-only and Cloud-assisted.

(c) With 25.85ms latency between DDAs in the same D2-ring, EF-dedup suffers more with larger D2-ring size but still outperforms Cloud-only and Cloud-assisted.

(d) For larger D2-ring size (i.e., more DDAs per ring), the deduplication ratio of EF-dedup approaches that of Cloud-only and Cloud-assisted.

(e) Larger similarity helps to improve throughput, EF-dedup outperform both Cloud-only and Cloud-assisted. Larger intra-group similarity helps Cloud-assisted more.

Fig. 4. Experimental clusters were created on Amazon EC2 and local OpenStack.

In Fig. 4(b), we evaluate the throughput performance of different strategies, as additional network latency between DDAs on the network edge and CDAs in the cloud is introduced using Netem. While all three strategies are negatively impacted by additional latency, EF-dedup still achieves significant throughput improvement, and its lead over Cloud-assisted strategy is maintained even under high latency. In particular, as the additional latency increases from 0 to 30ms, EF-dedup’s throughput drops by only 20.28%, compared to 43.33% for Cloud-assisted strategy. This is due to the fact that in EF-dedup, chunk/hash look-ups for distributed deduplication only generate network traffic between DDAs on network edge, making it more resilient to adverse network conditions at the cloud.

Fig. 4(c) compares the throughput performance of different strategies for varying D2-ring size. We use vdbench to create two groups of workload files. Within each group, any 2 files have deduplication ratio as 5. Across group, any 2 files have deduplication ratio as 2. Each DDA is assigned one 750 MB workload file. Since Cloud-assist and Cloud-only algorithms use global deduplication, consequently they have the same deduplication rate as 6.04. The overall deduplication ratio for EF-dedup is increasing with larger regional edge network size due to higher deduplication ratio but the increasing speed is decreasing. As a result, the throughput of Cloud-assist and Cloud-only is almost flat, but EF-dedup has growing throughput with a larger speed with small ring size below 6 nodes. For the smallest ring size as 2, the performance difference between EF-dedup and Cloud-assist is 35.74%, but this difference enlarges to 49.72%.

Fig. 4(d)

Figs. 4(e) and 4(f) evaluate the three strategies using synthetic files created by vdbench. We fix the network size as 20 DDAs and generate 20 synthetic files that belong to two different groups. More precisely, files within each group have a pre-fixed intra-group deduplication ratio of 2 or 5 (in Figs. 4(e) and 4(f), respectively), while files from different groups have an inter-group deduplication ratio varying from 2-10 in the evaluation. We show that EF-dedup achieves much higher throughput than both Cloud-only and Cloud-assist strategies. More importantly, the improvement is more substantial under high intra-group and low intra-group deduplication in Figs. 4(e). This is because by partitioning the DDAs into two D2-rings, EF-Dedup can effectively eliminate the high intra-group data redundancy (that exists within each D2-ring) via distributed deduplication, resulting in much higher throughput. Thus, to fully leverage the benefit of EF-Dedup, we need to intelligently partition the DDAs into D2-rings based on both network conditions and data similarity structure.

C. EF-dedup Micro-benchmarks

Shijing: Use term D2-rings to indicate 'clusters' in figures.

To demonstrate the effectiveness of the proposed EF-dedup strategy, we evenly distribute the 20 DDAs to 4 regional
edge networks (e.g. data centers, LANs) and use Netem to introduce extra latency for DDAs belonging to different data centers.

In Fig. 5(a), we show why we need to partition DDAs. We set 4 groups of DDAs where each group contains 5 DDAs, representing regional edge networks that are remote from each other. We use Netem to introduce latency among DDAs from different regional edge networks. We form 1, 2 and 4 D2-rings for comparison, and apply workloads in Fig. 4(d). When there is no latency, the throughput is growing with larger cluster size. This is because a larger cluster with higher deduplication ratio will increase the redundant data chunks that will not be transmitted again. However, a larger cluster means more peer-to-peer traffic which is sensitive to latency. Thus, as shown in Fig. 5(a), the throughput of 1 and 2 D2-rings decreases to 39.46% and 56.31% respectively of the ones without latency. This is because, with the increase of latency, the peer-to-peer traffic requires longer time and even traffic resending and then cause throughput decrease. For the 4 D2-rings partitioning, the throughput remains the same because there is no latency within the partitioning.

In Fig. 5(b) and Fig. 5(c), we show the trade off of deduplication ratio, throughput and network cost described in 2. Compared to 4 regional edge networks and 11 varying latencies in Fig. 5(a), we set 10 regional edge networks where each network includes 2 DDAs and introduce 4 typical latencies - 0, 25, 50 and 75ms. When there are more D2-rings (fewer nodes in each D2-ring), the network cost decreases dramatically, which is because inner D2-ring latencies cause network cost if 2 or more regional edge networks are included in the same D2-ring. Thus, increases the throughputs by 37.73%, 73.32%, and 94.43% while 25, 50, and 75ms latencies are added among regional edge networks. In no additional latency case, the throughput increases a little when the size of D2-rings enlarge. This is because when the size of D2-ring increase, the deduplication ratio increases to decrease large chunk transmission, although more peer-to-peer conversation traffic will decrease the throughput. In this situation, fewer large chunk transmission outweighs more peer-to-peer conversations.

To reduce the negative influence of latency on throughput, we propose arbitrary and balanced EDF-OP algorithms described in ??). We conduct experiments and simulations to compare the performance of EDF-OP algorithms with heuristic Network-Only and Dedup-Only algorithms. Network-only algorithm uses Greedy algorithm to do balanced or arbitrary D2-ring partitioning by minimizing the network cost; dedup-only algorithm partition D2-rings by searching the largest similarities; and EDF-OP algorithm minimizes $\sum_s U(P_s) + \alpha \sum_s V(P_s)$. In arbitrary algorithms, we fix the number of D2-rings and partition DDAs by minimizing overall cost; and in balanced algorithms, we make the rings have identical size. Arbitrary algorithms might have less cost, but at expense make the size of rings differ.
greatly from each other, which is not good for networks like data centers.

**Algorithm 4** Greedy Balanced Partitioning Algorithms

```plaintext
foreach cluster i
    foreach Z-node partition \( P_j \) in remaining nodes
        Network-Only: Calculate network cost \( V(P_j) \);
        Dedup-Only: Calculate storage cost \( U(P_j) \);
        Find \( P_{\min} \) with the smallest cost;
    end
    \( P_i = P_{\min} \);
    Remove Z nodes of partition \( P_i \): \( N \leftarrow N - P_i \);
end
```

**Algorithm 5** Arbitrary Partitioning Algorithm

```plaintext
foreach node i
    foreach cluster j
        Network-Only: Calculate cost \( V(P_j \cup \{i\}) - V(P_j) \);
        Dedup-Only: Calculate cost \( U(P_j \cup \{i\}) - U(P_j) \);
        Find \( P_{\min} \) with the smallest cost;
    end
    \( P_{\min} = P_{\min} \cup \{i\} \);
    Remove node i : \( N \leftarrow N - \{i\} \);
end
```

In Fig. 6(a) and Fig. 6(b), we show various performance of different methods of D2-ring partitioning. We set 10 regional edge networks where each network includes 2 DDAs and introduce latencies across networks where latencies have uniform distribution from 0 to 100ms. Sources used in Fig. 4(d) are randomly given to each DDA. \( \alpha \) is as 0.1. As shown in Fig. 6(a), Network-Only algorithm has 1.25-time aggregate cost and Dedup-Only algorithm has 1.3-time aggregate cost than EFD-OP. This is because EFD-OP will intelligently do trade-off to network and storage cost, which makes EFD-OP beats Network-Only by much less storage cost at expense of a little more network cost and similarly beats Dedup-Only algorithm. As result, shown in Fig. 6(b), EFD-OP saves 1792.36 MB storage space compared to Network-Only but has 7.24MB/s less throughput. Also, EDF-OP has 39MB/s more throughput than Dedup-Only but occupies 157.07MB storage.

**D. Simulations for EF-dedup deep dive**

1) **EF-dedup Performance for different flavors:** In this section, we simulate the performance of different partitioning algorithms with large scale(up to 100 DDAs), which is not easy for experiments. We simulate network cost and deduplication ratio performance of network-only, dedup-only algorithms and EF-dedup algorithm by using the characteristic vectors we obtained from Linux Kernel and Cassandra Source Codes(described in section ??). We set 3 chunk pools, indicating Linux only, Cassandra only and the overlap. Workload in each DDA has uniform distribution from 0 to 100GB. Chunk size for deduplication is 128KB. Network latency among DDAs has uniform distribution from 0 to 100ms.

![Graph showing overall cost varying with network size](image)

(a) Balanced Algorithms: EFD-OP has Smallest Aggregate Cost. This advantage Increases with More DDAs.

![Graph showing overall cost varying with number of DDAs](image)

(b) Arbitrary Algorithms: EFD-OP has Smallest Aggregate Cost. This advantage Increases with More DDAs. Arbitrary Algorithms Have Smaller Cost than Balanced Algorithms.

Fig. 7. Overall Cost with Varying Network Size

Fig. 7 show the aggregate, network and storage cost of 3 arbitrary and 3 balanced algorithms with varying network size and fixed \( \alpha \) as 0.001. Although the arbitrary partitioning EFD-OP algorithm(blue bars) has larger network cost than Network-Only algorithm(yellow bars) and larger storage cost than Dedup-Only algorithm(red bars) a little, its storage cost performance overwhelmingly beats Network-Only algorithm and network cost performance beats Dedup-Only algorithm. As a result, the EFD-OP algorithm win in the trade off performance - aggregation cost. This advantage is increasing when the network size is growing. This is because the partitioning options will increase with bigger network and make the benefits larger. When there are 100 nodes in the network, arbitrary EFD-OP algorithm has 62.63% & 45.64% less aggregate costs than arbitrary Network-Only & Dedup-Only algorithms respectively, balanced EFD-OP algorithm has 30.33% & 20.04% less aggregate costs than balanced Network-Only & Dedup-Only algorithms respectively. Arbitrary partitioning EFD-OP algorithm beats the balanced one since the balanced partitioning is included in arbitrary and has more limits to optimization.
To further optimize deduplication ratio. As an example, HYDRAstor [?] to HYDRAstor will perform more fine-grained deduplication. Other load balancing algorithms, and the servers involved in the process will perform more fine-grained deduplication to further optimize deduplication ratio. As an example, HYDRAstor [?] first deduplicates incoming data using larger chunk size, e.g., 64KB, then send intermediate results to multiple servers so that they can perform more fine-grained deduplication. ∑-dedup [?] exploits similarity for distributing super-chunks and utilizes locality to achieve faster index lookups. Other works in this area [?], [?] also takes a similar approach at a high level. However, unlike our systems, these works mostly focused on how to best utilize dedicated hardware allocated for backup.

Another body of work on improving chunk indexing process focused on improving memory utilization of a single host, and thereby reducing the needs for accessing slow disk drives. The goal has been achieved in various contexts, for instance, based on locality [?], [?], data similarity [?], [?] and/or relying on faster media such as SSDs [?].

Locality-based approaches are typically targeted for backup workloads. DDFS [?] identifies that many similar (or identical) files appears in a backup stream, often in the same order. To capture the locality of the data stream, the system maintains a data structure which logically aggregates related chunks as a group. Later, when a chunk belonging to the group is observed by the system, it prefetches related chunks to memory. Then the consecutive requests will be served much faster with very high probability, since they will not hit the disk drives. DDFS also uses a bloom filter [?] to quickly decide new chunks. Block locality caching [?] also exploits block locality existed in backup workloads, i.e., between old and new backup data, and devised a heuristic to find a probable block alignment. Sparse Indexing [?] brings a small fraction of the index database to local memory to facilitate the deduplication process. The system uses a locality-based sampling heuristics to score and decide which data it will bring to the memory. All of these systems are interested in reducing the frequency of disk accesses.

Other work steers the way of indexing chunk fingerprints based on the similarity observed by the deduplication sub-system [?], [?], [?], [?]. Both SAP [?] and Aronovich et al. [?] devised a method to compare pair-wise similarity, and based on the results, they explored trade-offs between access-efficiency (or throughput) and space efficiency. The systems such as SiLo [?] and ∑-dedup exploit both data similarity and locality. Both first use data similarity to decide bigger processing units and then apply mining technique to identify data locality, e.g., similar to the observation made in [?]. However, SiLo’s focus lies in optimizing local memory utilization and ∑-dedup optimizes dedicated backup infrastructure in a single datacenter.

Our work proposes a way to perform efficient chunk indexing and lookup for data deduplication in a distributed environment, i.e., rather than relying on local memory space. Our problem setting is different from most work in the area of clustered deduplication. For instance, in our edge cloud setting, minimizing WAN bandwidth usage is important. So our design objective includes to maximally utilize available...
resources in each local premise, construct a shared storage space for indexing chunk fingerprints, which can be co-located with the clients. Moreover, our way of modeling deduplication process and partitioning data distribution of the indexes are significantly different from the existing similarity-based deduplication systems. need to be revisited later.

VII. CONCLUSION