MPCCache: Privacy-Preserving Multi-Party Cooperative Cache Sharing at the Edge

Duong Tung Nguyen and Ni Trieu(✉)

Arizona State University, Tempe, AZ, USA
{duongnt,nitrieu}@asu.edu

Abstract. We present MPCCache, an efficient **M**ulti-**P**arty **C**ooperative **Cache** sharing framework, which allows multiple network operators to determine a set of common data items with the highest access frequencies to be stored in their capacity-limited shared cache while guaranteeing the privacy of their individual datasets. The technical core of our MPCCache is a new construction that allows multiple parties to compute a specific function on the intersection set of their datasets, without revealing both the private data and the intersection itself to any party.

We evaluate our protocols to demonstrate their efficacy and practicality. The numerical results show that MPCCache scales well to large datasets and achieves a few hundred times faster compared to a baseline scheme that optimally combines existing MPC protocols.

1 Introduction

The explosive growth of data traffic due to the proliferation of wireless devices and bandwidth-hungry applications leads to an ever-increasing capacity demand across wireless networks to enable scalable wireless access with high quality of service (QoS). This trend will likely continue for the near future due to the emergence of new applications like augmented/virtual reality, 4K/8K UHD video, and tactile Internet [13]. Thus, it is imperative for mobile operators to develop cost-effective solutions to meet the soaring traffic demand and diverse requirements of various services in the next generation communication network.

Enabled by the drastic reduction in data storage cost, edge caching has appeared as a promising technology to tackle the aforementioned challenges in wireless networks [3]. In practice, many users in the same service area may request similar content such as highly-rated Netflix movies. Furthermore, most user requests are associated with a small amount of popular content. Hence, by proactively caching popular content at the network edge (e.g., at base stations, edge clouds) in advance during off-peak times, a portion of requests during peak hours can be served locally right at the edge instead of going all the way through the mobile core and the Internet to reach the origin servers. The new edge caching paradigm can significantly reduce duplicate data transmission, alleviate the backhaul capacity requirement, mitigate backbone network congestion, increase network throughput, and improve user experience [1,3,13,37].

I. Eyal and J. Garay (Eds.): FC 2022, LNCS 13411, pp. 80–99, 2022.
https://doi.org/10.1007/978-3-031-18283-9_5

Motivation. With edge caching, the advantages brought by cooperation become clear. Each operator can maintain a private cache and share a shared cache with others. Although the benefits of edge caching have been studied extensively in the previous literature along with many real-world deployments [1,3,37], most of the existing works on cooperative edge caching consider cooperation among edge caches owned by a single operator only [27,37,38]. The potential of cache cooperation among multiple operators has been overlooked. For cooperative cache sharing, the data privacy of individual Telcos is important. For example, if TelcoA knows the access pattern of subscribers of TelcoB, TelcoA can learn characteristics of TelcoB's subscribers and design incentive schemes and services to attract these subscribers to switch to TelcoA. Therefore, it is imperative to study various mechanisms that provide the benefits of cache sharing without compromising privacy.

Contributions. We introduce an MPCCache scheme to tackle the cooperative content caching problem at the network edge where multiple semi-honest parties (i.e., network operators) can jointly cache common data items in a shared cache. The problem is to identify the set of common items with the highest access frequency to be cached in the shared cache while respecting the privacy of each individual party. To the best of our knowledge, we are among the first to realize and formally examine the multi-party cooperative caching problem by exploiting the non-rivalry of cached data items, and tackle this problem through the lens of secure multi-party computation. We introduce an efficient construction that outputs only the result of a specific function computed securely on the intersection set, (i.e., find k best items in the intersection set) without revealing the private data of individual parties as well as the intersection itself to any party, and works for the multi-party setting with more than two parties. In addition, we propose an efficient top-k algorithm that achieves an approximate $\frac{\log^2(m)}{\big(\log(k)+2\big)\log(k)}\times$ improvement compared with the prior top-k algorithms, where m is the size of the dataset.

We demonstrate the practicality of our protocol with experimental numbers. For instance, for the setting of 8 parties each with a data-set of 2^{16} records, our decentralized protocol requires 5 min to compute k-priority common items for $k = 2^8$. We also propose an optimized server-aid MPCCache construction, which is scalable for large datasets and a number of parties. With 16 parties, each has 2^{20} records, our optimized scheme takes only 8 min to compute the k-priority common items for $k = 2^8$. MPCCache aims at proactive caching where caches are refreshed periodically (e.g., hourly). Therefore, the running time of MPCCache is practical in our application.

In addition to cooperative cache sharing as our main motivation, we believe that the proposed techniques can find applications in other areas as well.

2 Related Work and Technical Overview of MPCCache

Consider a single party with a set of items S. Each item includes an identity x (i.e., a file name, a content ID) and its associated value v. For each set S,

PARAMETERS: n parties $P_{i\in[n]}$, each has m_i items, a threshold k, where k is much smaller than the intersection size.
FUNCTIONALITY:
- Wait for an input $S_i = \{(x_1^i, v_1^i), \ldots, (x_{m_i}^i, v_{m_i}^i)\} \subset (\{0,1\}^\kappa, \{0,1\}^\theta)$ from P_i
- Let $I = \bigcap_{i\in[n]}\{x_1^i, \ldots, x_{m_i}^i\}$ to be the intersection set. For each $x^\star \in I$, compute the sum $v^\star$ of associated values, i.e., $v^\star = \sum_{i=1}^n v_{j_i}^i$ where $(x^\star, v_{j_i}^i) \in S_i$
- Give parties $\{x_1^\star, \ldots, x_k^\star\}$ where $v_1^\star, \ldots, v_k^\star$ are k largest numbers among $v_1^\star, \ldots, v_{|I|}^\star$.

Fig. 1. The MPCCache functionality

an element (x, v) is said to belong to a set of k-*priority* elements of S if its associated value v is one of the k-largest values in S. Note that the value of each content item may represent the number of predicted access frequency of the content or the benefit (valuation) of the operator for the cached content. Each network operator has its own criteria to define the value for each content that can be stored in the shared edge cache space. How to define the value for each content is beyond the scope of this work. In this work, we assume that the parties are truthful by using their true valuations for each content item in their databases. It is because the access frequency of each party to each cached file is measurable and known. Additionally, some economic penalty schemes can be used to enforce truthfulness as mentioned in the full version of the paper [25].

Since the cache is shared among the operators, they would like to store only common content items in the cache. Here, a common item refers to an item (based on identity) that is owned by every party. The common items with the highest values will be placed in the shared cache. The value of a common item is defined as the sum of the individual values of the operators for the item. Concretely, we consider the cooperative caching problem in the multi-party setting where each party P_i has a set $S_i = \{(x_1^i, v_1^i), \ldots, (x_{m_i}^i, v_{m_i}^i)\}$. Without loss of generality, we assume that all parties have the same set size m. An item $(x^\star, v^\star)$ is defined to belong to the set of the k-*priority common* elements if it satisfies the two following conditions: (1) $x^\star$ is the *common* identity of all parties; (2) $(x^\star, v^\star)$ are the k-*priority* elements of $S^\star = \{(x_1^\star, v_1^\star), \ldots, (x_{|I|}^\star, v_{|I|}^\star)\}$, where $v_i^\star$ is the sum of the values associated with these common identities from each party, and $I = \bigcap_{i\in[n]}\{x_1^i, \ldots, x_{m_i}^i\}$ is the intersection set with its size $|I|$. In the setting, we consider the input datasets of each P_i contain proprietary information, thus none of the parties are willing to share its data with the other. We describe the ideal functionality of MPCCache in Fig. 1. For simplicity, we remove under-script of the common item $x^\star$ and clarify that a pair $(x^\star, v_{j_i}^i) \in S_i$ belongs to P_i.

A closely related work to MPCCache is a private set intersection (PSI). Recall that the functionality of PSI enables n parties with respective input sets $X_{i\in[n]}$ to compute the intersection itself $\bigcap_{i\in[n]} X_i$ without revealing any information about the items which are not in the intersection. However, MPCCache requires to evaluate a top-K computation on the top of the intersection $\bigcap_{i\in[n]} X_i$ while also keeping the intersection secret from parties. The work [8,21,29,32] proposed optimized circuits for computing on the intersection by deciding which items of the parties need to be compared. However, their constructions only work for the two-party setting. Most of the existing multi-party PSI constructions [10,17,20,24,33] output the intersection itself. Only very few works [18,23] studied some specific functions on the intersection. While [18] does not deal with the intersection set of all parties (in particular, an item in the output set in [18] *is not necessarily a common item of all parties*), [23] finds common items with the highest preference (rank) among all parties. [23] can be extended to support MPCCache which is a general case of the rank computation. However, the extended protocol is very expensive since if an item has an associated value v, [23] represents the item by replicating it v times. For ranking, their solution is reasonable with small v but for our MPCCache it is not suitable since v can be a very large value. We describe a detailed discussion in the full version of the paper [25]. The work of [31] proposes MPCircuits, a customized MPC circuit. One can extend MPCircuits to identify the secret share of the intersection and use generic MPC protocols to compute a top-k function on the secret-shared intersection set. However, the number of secure comparisons inside MPCircuits is large and depends on the number of parties. A concurrent and independent work by Chandran et al. [7] is the state-of-the-art multi-party circuit-PSI, but only supports a weaker adversary, who may corrupt at most $t < n/2$ the parties. Moreover, in terms of theoretical complexity comparisons, [7] is expensive than ours. We explicitly compare our proposed MPCCache with the MPCircuits and [7] in Sect. 6.3.

Our decentralized MPCCache construction contains two main phases. The first one is to obliviously identify the common items (i.e., items in the intersection set) and aggregate their associated values of the common items in the multi-party setting. In particular, if all parties have the same $x^\star$ in their set, they obtain secret shares of the sum of the associated values $v^\star = \sum_{i=1}^{n} v^i_{j_i}$ where $(x^\star, v^i_{j_i}) \in S_i$. Otherwise, $v^\star$ equals to zero and it should not be counted as a k-priority element. A more detailed overview of the approach is presented in Sect. 4. It is worth mentioning that the first phase does not leak the intersection set to any party. The second phase takes these secret shares which are either the zero value or the correct sum of the associated values of common items, and outputs k-priority items. To privately choose the k-priority elements that are secret shared by n parties, one could study top-k algorithms.

In MPC setting, a popular method for securely finding the top-k elements is to use an oblivious sort (i.e., parties jointly sort the dataset in decreasing order of the associated values, and pick the k largest values). The most practical algorithm is Batcher's network [4], which computational and communication complexity are $O(m\log^2(m))$ and $O(\ell m\log^2(m))$, respectively, where m is the size of the dataset and ℓ is the bit-length of the element (see the full version of the paper [25] for more detail). To output the index of the k largest values, we also need to keep track of their indexes, therefore, the total communication complexity of oblivious Batcher's network is $O((\ell + \log(m))m\log^2(m))$. Another approach to compute k-priority elements is to use an oblivious heap that allows to get a maximum element from the heap (ExtractMax). This solution requires to call ExtractMax k times, which leads to a number of rounds of the interaction of at least $O(k \log(m))$.

In MPCCache, the size of an edge cache k is usually much smaller than the size of the dataset m. In addition, it is also much smaller than the caching facility at the core of the network operator. Since we are motivated by applications where $k \ll m$, we propose a new protocol with computational and communication overhead of $O(m \log^2(k))$ of secure comparisons and $O((\ell + \log(m))m \log^2(k))$ bits, respectively. Our protocol requires $O(\log(m))$ rounds. Concretely, we show an approximate $\frac{\log^2(m)}{\left(\log(k)+2\right)\log(k)} \times$ improvement compared with the prior work.

Recently, [9] presents an *approximate* top-K selection with complexity of $O(m+k^2)$ comparisons and $O((\ell+\log(m))(m+k^2)$ bits. One could integrate their algorithm in the second phase of our scheme to achieve better performance. In applications where *exact* top-K selection is required, our k-priority is preferable.

Our decentralized protocol supports the full corrupted majority, which means that if any subset of parties is corrupted, they learn nothing except the protocol output. In this paper, we also present the optimization for MPCCache in the non-colluding semi-honest setting in which we assume to know two non-colluding parties. This model can be considered as the server-aided model where clients obliviously distribute (secret share) their private database to two non-colluding servers. Our optimized server-aided MPCCache construction achieves almost the same cost as that of our two-party decentralized protocol.

3 Cryptographic Preliminaries

In this work, the computational and statistical security parameters are denoted by κ, λ, respectively. We use $[.]$ notation to refer to a set, and $[i, j]$ to denote the set $\{i, \ldots, j\}$. The additive secret sharing of a value x is defined as $[\![x]\!]$.

Secret Sharing. To additively secret share $[\![x]\!]$ an ℓ-bit value x of the party P_i to other parties, he first chooses $x^i \leftarrow \mathbb{Z}_{2^\ell}$ uniformly at random such that $x = \sum_{j=1}^{n} x^j \mod 2^\ell$, and then sends each x^j to the party P_j. For ease of composition, we omit the mod. To reconstruct an additive shared value $[\![x]\!]$, all parties P_j sends $[\![x]\!] = x^j$ to the party P_i, who locally reconstructs the secret value by computing $x \leftarrow \sum_{i=1}^{n} x^j$. In this work, we also use Boolean sharing in the binary field. Boolean sharing can be seen as additive sharing in the field $\mathbb{Z}_2$.

Oblivious Key-Value Store (OKVS). An OKVS [14] is a data structure in which a sender, holding a set of key-value mapping $\mathit{\Gamma} = \{(k_i, v_i), i \in [n]\}$ with pseudo-random v_i, wishes to give that mapping over to a receiver who can evaluate the mapping on any input but without revealing the keys k_i. Formally, an OKVS consists of two algorithms: $\mathsf{Encode}(\mathit{\Gamma}) \rightarrow \mathcal{T}$ is a randomized algorithm that takes as input a set of n key-value pairs $\mathit{\Gamma} = \{(k_i, v_i)_{i \in [n]}\}$ from the domain $\mathcal{K} \times \mathcal{V}$, outputs a table $\mathcal{T}$; and $\mathsf{Decode}(k, \mathcal{T}) \rightarrow v$ is a deterministic algorithm that takes as input a table $\mathcal{T}$, a key k and outputs a value v.

The correctness of the OKVS is that if for all key-value pairs $A \subseteq \mathcal{K} \times \mathcal{V}$ with distinct keys and pseudo-random values, $\mathsf{Encode}(A) = \mathcal{T}$ and $(k, v) \in A$ then $\mathsf{Decode}(k, \mathcal{T}) = v$. An OKVS is secure if the values v_i are chosen uniformly then the output of Encode hides the choice of the keys k_i.

Garbled Circuit. An ideal functionality GC [5,16,36] is to take the inputs x_i from party P_i, and computes a function f on them without revealing the parties' inputs. We use Yao [36] and BMR-style protocols [5,6] for two-party and multi-party GC, respectively. In our protocol, we use f as "less than" and "equality" where inputs are secretly shared amongst all parties. For example, a "less than" GC takes the parties' secret shares $[\![x]\!]$ and $[\![y]\!]$ as input, and output the shares of 1 if $x < y$ and 0 otherwise. We denote the GC by $[\![z]\!] \leftarrow \mathcal{GC}([\![x]\!], [\![y]\!], f)$.

Oblivious Sort and Merge. The main building block of the sorting algorithm is $\mathsf{Compare\text{-}Swap}$ operation that takes the secret shares of two values x and y, then compares and swaps them if they are out of order. It is typical to measure the complexity of oblivious sort/merge based on the number of $\mathsf{Compare\text{-}Swap}$.

Oblivious Sort: We denote the oblivious sorting by $\{[\![x_i]\!]_{i\in[m]}\} \leftarrow \mathcal{F}_{\mathsf{obv\text{-}sort}}(\{[\![x_i]\!]_{i\in[m]}\}$ which takes the secret share of m values and returns their refresh shares in which all $x_{i\in[m]}$ are sorted in decreasing order. As discussed in [25], Batcher's network for oblivious sort requires $\frac{1}{4}m\log^2(m)$ $\mathsf{Compare\text{-}Swap}$ operations.

Oblivious Merge: Given two sorted sequences, each of size m, we also need to merge them into a sorted array, which is part of the Batcher's oblivious merge sort. It is possible to divide the input sequences into their odd and even parts, and then combine them into an interleaved sequence. This oblivious merge requires $\frac{1}{2}m\log(m)$ $\mathsf{Compare\text{-}Swap}$ operations and has a depth of $\log(m)$. We denote the oblivious merge by $\{[\![z_1]\!], \ldots, [\![z_{2m}]\!]\} \leftarrow \mathcal{F}_{\mathsf{obv\text{-}merge}}(\{[\![x_1]\!], \ldots, [\![x_m]\!]\}, \{[\![y_1]\!], \ldots, [\![y_m]\!]\})$.

4 Our Decentralized MPCCache Construction

Recall that our MPCCache construction contains two main parts. The first phase allows parties to securely generate shares of the sum of the associated values under a condition. More precisely, if all parties have x in their sets then the sum of their obtained shares is equal to the sum of the associated values for the common x. Otherwise, the sum of the shares is zero. These shares are forwarded as input to the second phase, which ignores the zero sum and returns only k-priority common items. For the second phase, we first present the $\mathcal{F}_{\mathsf{k\text{-}prior}}$ functionality of computing *k-priority* elements in Fig. 2, and use it as a black box in our MPCCache construction. We describe our $\mathcal{F}_{\mathsf{k\text{-}prior}}$ construction in Sect. 4.3.

4.1 A Special Case of Our First Phase

We start with a special case. Suppose that each party $P_{i\in[n]}$ has only one item (x^i, v^i) in its set S_i. Our first phase must satisfy the following conditions:

PARAMETERS: Set size m, and n parties
FUNCTIONALITY:
- Wait for secret shares $\{[\![v_1]\!], ..., [\![v_m]\!]\}$ from the i^{th} party.
- Give all parties k indexes $\{i_1, \ldots, i_k\}$ such that $\{v_{i_1}, \ldots, v_{i_k}\}$ are largest values among $\{v_1, ..., v_m\}$.

Fig. 2. The k-priority functionality ($\mathcal{F}_{\mathsf{k\text{-}prior}}$)

(1) If all x^i are equal, the parties obtain secret shares of the sum of the associated values as $v^\star = \sum_{i=1}^n v^i$.
(2) Otherwise, the parties obtain secret shares of zero.
(3) The protocol is secure in the semi-honest model, against any number of corrupt, colluding parties.

The requirement (3) implies that all corrupt parties should learn nothing about the input of honest parties. To satisfying (3), the protocol must ensure that parties do not learn which of the cases (1) or (2) occurs.

We assume that there is a leader party (say P_1) who interacts with other parties to output (1). The protocol works as follows. For (x^i, v^i), $P_{i\neq 1}$ chooses a secret $s^i \in \{0,1\}^\theta$ uniformly at random, and defines $w^i \overset{\mathsf{def}}{=} v^i - s^i$ (for ease of composition we omit the mod). He then computes a one-time pad as $\mathsf{OTP}(x^i, w^i) = x^i \oplus w^i$ (for simplicity, we assume that the domain size of x^i and w^i are equal; it is also possible to use $H(x^i)$ instead of the original item x^i, where $H : \{0,1\}^\star \to \{0,1\}^\star$ is a collision-resistant hash function). The $P_{i\neq 1}$ then sends the ciphertext to the leader P_1. Using his item x^1, the P_1 decrypts the received ciphertext and obtains w^i if $x^1 = x^i$, random otherwise. Clearly, if all parties have the same x^1, P_1 receives $w^i = v^i - s^i$ from $P_{i\neq 1}$. Now, P_1 computes $s^1 \overset{\mathsf{def}}{=} v^1 + \sum_{i=2}^n w^i$. It easy to verify that $\sum_{i=1}^n s^i = (v^1 + \sum_{i=2}^n w^i) + \sum_{i=2}^n s^i = v^1 + \sum_{i=2}^n (w^i + s^i) = \sum_{i=1}^n v^i = v^\star$. By doing so, each P_i has an additive secret share s^i of $v^\star$ as required in (1).

In case that not all x^i are equal, the sum of all the shares $\sum_{i=1}^n s^i$ is a random value since P_1 receives a random (incorrect) w^i from some party/parties. To satisfy (2), we use GC to turn the random sum $\sum_{i=1}^n s^i$ to zero. However, for (3), the random sum and the correct sum are indistinguishable from the view of all parties. One might make use of GC by computing n equality comparisons to check whether all x^i is equal. If yes, the circuit gives refreshed shares of the correct sum, otherwise shares of zero. This solution requires $O(n)$ equality comparisons inside MPC. We aim to minimize the number of equality tests.

We improve the above solution using zero-sharing [2,20,22]. An advantage of the zero-sharing is that the party can non-interactively generate a Boolean share of zero after a one-time setup. Let's denote the zero share of P_i to be z^i. We have $\bigoplus_{i=1}^n z^i = 0$. Similar to the protocol described above to achieve (1): Instead of (x^i, v^i), the P_i uses (x^i, z^i) as input, and receives a Boolean secret share t^i. If all x^i are equal, the XOR of all obtained shares is equal to the XOR of all associated values z^i. In other words, $\bigoplus_{i=1}^n t^i = \bigoplus_{i=1}^n z^i = 0$. Otherwise, $\bigoplus_{i=1}^n t^i$

is random. These obtained shares are used as an `if` condition to output either (1) or (2). Concretely, parties jointly execute a garbled circuit to check whether $\bigoplus_{i=1}^{n} t^i = 0$. If yes (*i.e.* parties have the same item), the circuit re-randomizes the shares of $v^\star$, otherwise, generates the shares of zero. The zero-sharing based solution requires only one equality comparison inside MPC.

We now describe a detailed construction to generate zero-sharing [20] and how to compute t^i, w^i more efficiently.

a) Zero-sharing key setup: one key is shared between every pair of parties. For example, the key k_{ij} is for a pair (P_i, P_j) where $i, j \in [n], i < j$. It can be done as P_i randomly chooses $k_{i,j} \leftarrow \{0,1\}^\kappa$ and sends it to P_j. Let's denote a set of the zero-sharing keys of P_i as $K_i = \{k_{i,1}, \ldots, k_{i,(i-1)}, k_{i,(i+1)}, \ldots, k_{i,n}\}$.
b) Generating zero share: Given a PRF $F : \{0,1\}^\kappa \times \{0,1\}^* \rightarrow \{0,1\}^*$, a set of keys K_i and a value x, each P_i locally computes a zero share of x as $z^i = \bigoplus_{j=1}^{n} F(k_{i,j}, x)$. Clearly, each term $F(k_{i,j}, x)$ appears exactly twice in the expression $\bigoplus_{i=1}^{n} z^i$. Thus, $\bigoplus_{i=1}^{n} z^i = 0$. We define $f^{\mathsf{z}}(K_i, x) \overset{\mathsf{def}}{=} \bigoplus_{j=1}^{n} F(k_{ij}, x)$ for P_i to generate the zero share of x.
c) Computing s^1 and t^1: the $P_{i\neq 1}$ chooses random s^i and t^i. For an input (x^i, v^i) and a zero share $z^i \leftarrow f^{\mathsf{z}}(K_i, x^i)$, he computes $w^i \overset{\mathsf{def}}{=} v^i - s^i$ and $y^i \overset{\mathsf{def}}{=} z^i \oplus t^i$ and sends the one-time pad $\mathsf{OTP}(x^i, y^i || w^i)$ to the leader P_1 (assume that the length of x^i and $y^i || w^i$ are equal). Using his item x^1 as a decryption key, P_1 obtains the correct $y^i || w^i$ if $x^1 = x^i$, random otherwise. P_1 computes $s^1 \overset{\mathsf{def}}{=} v^1 + \sum_{i=2}^{n} w^i$ and $t^1 \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^{n} y^i) \oplus z^1$. At this point, each P_i has secret shares s^i and t^i such that $\sum_{i=1}^{n} s^i = v^\star$ and $\bigoplus_{i=1}^{n} t^i = 0$ if all x^i are equal.

4.2 A General Case of Our First Phase

So far, we only consider the simple case where each party has only one item. In this section, we show how to efficiently extend our protocol to support the general case where $m > 1$. At the high-level idea, we use hashing scheme to map the common items into the same bin and then reply on OKVS to compress each bin into a share so that the parties can evaluate $\mathsf{MPCCache}$ bin-by-bin efficiently.

Similar to many PSI constructions [19,28], we use two popular hashing schemes: Cuckoo and Simple. The leader P_1 uses Cuckoo hashing [26] with $\widetilde{k} = 3$ hash functions to map his $\{x_1^1, \ldots, x_m^1\}$ into $\beta = 1.27m$ bins. He then pads his bin with dummy items so that each bin contains exactly one item. This step is to hide his actual Cuckoo bin size. On the other hand, each $P_{i\neq 1}$ use the same $\widetilde{k}$ Cuckoo hash functions to place its $\{x_1^i, \ldots, x_m^i\}$ into β bins (so-called Simple hashing), each item is placed into $\widetilde{k}$ bins with high probability. The $P_{i\neq 1}$ also pads his bin with dummy items so that each bin contains exactly $\gamma = 2\log(m)$ items. According to [12,28], the parameters $\beta, \widetilde{k}, \gamma$ are chosen so that with the probability $1-2^{-\lambda}$ every Cuckoo bin contains at most one item and no Simple bin contains more than γ items. More detail is described in the full version of the paper [25].

For each bin b^{th}, P_1 and $P_{i\neq 1}$ can run a special-case protocol described in Sect. 4.1. In particular, let $B_i[b]$ denote the set of items in the b^{th} bin of P_i. All parties locally generate zero shares $z^i_j \leftarrow f^{\mathsf{z}}(K_i, x^i_j)$. The $P_{i\neq 1}$ locally chooses random values s^i_b and t^i_b. For each $(x^i_j, v^i_j) \in B_i[b]$, $P_{i\neq 1}$ computes $w^i_j \overset{\mathsf{def}}{=} v^i_j - s^i_b$ and $y^i_j \overset{\mathsf{def}}{=} z^i_j \oplus t^i_b$ and sends the one-time pad ciphertext $\mathsf{OTP}(x^i_j, y^i_j || w^i_j)$ to the leader P_1. Using his item $x^1_b \in B_1[b]$ as a decryption key, P_1 obtains $\hat{y}^i_j || \hat{w}^i_j$ which equals $y^i_j || w^i_j$ if $x^1_b = x^i_j$, random otherwise. Since there are γ values $\hat{y}^i_j || \hat{w}^i_j$, each for a pair in $B_i[b]$, obtained from $P_{i\neq 1}$, the P_1 has γ^{n-1} possible ways to choose $j_i \in [\gamma]$ and compute his share $s^1_b \overset{\mathsf{def}}{=} v^1_b + \sum_{i=2}^n \hat{w}^i_{j_i}$ and $t^1_b \overset{\mathsf{def}}{=} \bigoplus_{i=2}^n \hat{y}^i_{j_i} \oplus z^1_b$. Thus, this solution requires γ^{n-1} equality comparisons to check all combinations of whether $\bigoplus_{i=1}^n t^i_b = 0$ to determine whether x^1_b is common.

To improve the above computation, we rely on an OKVS data structure in order that P_1 learns from $P_{i\neq 1}$ only one pair $\{\hat{y}^i, \hat{w}^i\}$ per bin, instead of γ pairs per bin. More precisely, for each bin b, the party $P_{i\neq 1}$ creates a set of points $\Gamma^i_b = \{(x^i_j, y^i_j || w^i_j) \mid x^i_j \in B_i[b]\}$, encodes it as $\mathsf{Encode}(\Gamma^i_b) \rightarrow \mathcal{T}^i_b$ and sends the OKVS table $\mathcal{T}^i_b$ to the leader P_1. Thanks to the oblivious property of OKVS, we no longer need the one-time pad encryption. Using x^1_b, the P_1 decodes $\mathcal{T}^i_b$ and obtains $\hat{y}^i_b || \hat{w}^i_b \leftarrow \mathsf{Decode}(x^1_b, \mathcal{T}^i_b)$. Note that, if $x^1_b \in B_{i\neq 1}[b]$, $\hat{y}^i_b || \hat{w}^i_b$ equals to a $y^i_{j_i} || w^i_{j_i}$ that was encoded in $\mathcal{T}^i_b$, and otherwise, random.

In summary, if all parties have x^1_b in their b^{th} bin, the leader P_1 receives $\hat{w}^i_b = v^i_{j_i} - s^i_b$ and $\hat{y}^i_b = z^i_j \oplus t^i_b$ from the corresponding OKVS execution involving $P_{i\neq 1}$. The leader computes $s^1_b \overset{\mathsf{def}}{=} v^1_b + \sum_{i=2}^n \hat{w}^i_b$. If all parties have x^1_b, we have $\sum_{i=1}^n s^i_b$ is equal to the sum of the associated values corresponding with the identity x^1_b. Similarly, when defining $t^1_b \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n \hat{y}^i_b) \oplus z^1_b$, we have $\bigoplus_{i=1}^n t^i_b = 0$ if all parties have x^1_b. Consider a case that some parties $P_{i\neq 1}$ might not hold the item $x^1_b \in B_1[b]$ that P_1 has, the corresponding OKVS with these parties gives P_1 random $\hat{y}^i_b || \hat{w}^i_b$. Thus $t^1_b \overset{\mathsf{def}}{=} (\bigoplus_{i=2}^n \hat{y}^i_b) \oplus z^1_b$ is random, so is $\bigoplus_{i=1}^n t^i_b$.

Similar to Sect. 4.1, we use GC to check whether $\bigoplus_{i=1}^n t^i_b = 0$ for the bin b, and outputs either refreshed shares of $\sum_{i=1}^n s^i_b$ or shares of zero. Since P_1 only has one s^1_b, the protocol only needs to execute one comparison circuit per bin, thus the number of equality tests needed is linear in the number of the bins.

Even though $P_{i\neq 1}$ uses the same offset s^i_b, t^i_b per bin, all w^i_j and y^i_j are random (assume that v^i_j is randomly distributed). In addition, the OKVS only gives P_1 one pair per bin. Therefore, as long as the OKVS used is secure, so is our first phase of MPCCache construction. We formalize and prove secure our first phase which is presented, together with proof of our MPCCache security in Sect. 4.4.

4.3 Our Second Phase: *k-priority* Construction

In this section, we measure the complexity of our *k-priority* protocol based on the number of secure Compare-Swap operations. As discussed in Sect. 2, one could use oblivious sorting to sort the input set and then take the indexes of k biggest values. This approach requires about $\frac{1}{4}m\log^2(m)$ Compare-Swap operations and the depth of $\log(m)$. In the following, we describe our simple construction which costs $\left(\frac{1}{4}\log(k) + \frac{1}{2}\right)m\log(k) - \frac{1}{2}k\log(k)$ Compare-Swap with the same depth. The proposed algorithm achieves an approximate $\frac{\log^2(m)}{\left(\log(k)+2\right)\log(k)}\times$ improvement.

PARAMETERS:
- Set size m, a bit-length θ, security parameter λ, and n parties $P_{i\in[n]}$
- A zero-sharing key setup, GC, and k-$\mathsf{priority}$ primitives
- An OKVS data structure with Encode and Decode algorithms.
- A Cuckoo and Simple hashing with 3 hash functions, β bins, and max bin size γ.

INPUT OF PARTY $P_{i\in[n]}$: A set $S_i = \{(x^i_1, v^i_1), \ldots, (x^i_m, v^i_m)\} \subset \big(\{0,1\}^*, \{0,1\}^\theta\big)^m$

PROTOCOL:

I. **Pre-processing**.
1. Each party P_i interacts with other parties $\{P_1, \ldots, P_{i-1}, P_{i+1}, P_n\}$ to generate a zero-sharing key K_i and locally computes zero shares as $z^i_j \leftarrow f^z_i(K_i, x^i_j), \forall j \in [m]$.
2. A leader P_1 hashes $\{x^1_1, \ldots, x^1_m\}$ into β bins using the Cuckoo hashing scheme. Let $B_1[b]$ denote the item in the bth bin (or a dummy item if this bin is empty).
3. Each party $P_{i\in[2,n]}$ hashes items $\{x^i_1, \ldots, x^i_m\}$ into β bins using Simple hashing. Let $B_i[b]$ denote the set of items in the b^{th} bin of this party.

II. **Online**.
1. For each bin $b \in [\beta]$:
   a) Each party $P_{i\in[2,n]}$ chooses $t^i_b \leftarrow \{0,1\}^{\lambda+\log(n)}$ and $s^i_b \leftarrow \{0,1\}^\theta$ at random, and generates a set of key-value pairs $\Gamma^i_b = \{\big(x^i_j, y^i_j||w^i_j\big) \mid x^i_j \in B_i[b]\}$ where $y^i_j \overset{\text{def}}{=} z^i_j \oplus t^i_b$ and $w^i_j \overset{\text{def}}{=} v^i_j - s^i_b$. The party then pads Γ^i_b with dummy pairs to γ.
   b) Each party $P_{i\in[2,n]}$ encodes Γ^i_b as $\mathsf{Encode}(\Gamma^i_b) \rightarrow \mathcal{T}^i_b$ and sends $\mathcal{T}^i_b$ to P_1 who computes $\mathsf{Decode}(x^1_b, \mathcal{T}^i_b)$ and obtains $\hat{y}^i_b||\hat{w}^i_b$. Note that $\hat{y}^i_b = z^i_{j_i} \oplus t^i_b$ and $\hat{w}^i_b = v^i_{j_i} - s^i_b$ for $x^1_b = x^2_{j_2} = \ldots = x^n_{j_n}$. Otherwise, $\hat{y}^i_b, \hat{w}^i_b$ are random.
   c) P_1 computes $t^1_b \overset{\text{def}}{=} (\bigoplus_{i=2}^n \hat{y}^i_b) \oplus z^1_b$ and $s^1_b \overset{\text{def}}{=} v^1_b + \sum_{i=2}^n \hat{w}^i_b$ where z^1_b and v^1_b are zero share and the associated value corresponding to x^1_b, respectively.
   d) Parties jointly invoke a GC instance:
      - Input from P_i is t^i_b and s^i_b.
      - Output to P_i is an additive share $[\![u_b]\!]$ where $u_b = \sum_{i=1}^n s^i_b$ if $\bigoplus_{i=1}^n t^i_b = 0$, otherwise $u_b = 0$.

      Note that if x^1_b is common, u_b is equal to the sum of its associated values of the common item identity x^1_b.
2. Parties invoke a k-$\mathsf{priority}$ functionality with input $[\![u_b]\!], \forall b \in [\beta]$, and obtain k indexes of the k-$\mathsf{priority}$ common identities.

Fig. 3. Our decentralized $\mathsf{MPCCache}$ construction.

The main idea of our construction is that parties divide the input set into $\lceil \frac{m}{k} \rceil$ groups, each has k items except possibly the last group which may have less than k items (without loss of generality, we assume that m is divisible by k). Parties then execute an oblivious sorting invocation within each group to sort these values of this group in decreasing order. Unlike the recent work [9] for *approximate* top-K selection where it selects the maximum element within each group for further computation, we select the top-K elements of two neighbor groups. Concretely, the oblivious merger is built on top of each two sorted neighbor groups. We select only a set of the top-K elements from each merger and recursively merge two selected sets until reaching the final result.

Sorting each group requires $\frac{1}{4}k\log^2(k)$ $\mathsf{Compare\text{-}Swap}$ invocations, thus, for $\frac{m}{k}$ groups the total $\mathsf{Compare\text{-}Swap}$ operations needed is $\frac{m}{k}\big(\frac{1}{4}k\log^2(k)\big)$. The oblivious odd-even mergers are performed in a binary tree structure. The merger of two sorted neighbor groups, each has k items, is computed at each node of the tree. Unlike the sorting algorithm, we truncate this resulted array, maintain the secret shares of only k largest sorted numbers among these two groups, and throw out the rest of k numbers. By doing so, instead of $2k$, only k items are forwarded to the next odd-even merger. The number of Compare-Swap required for each merger does not blow up, and is equal to $\frac{1}{2}k\log(k)$. After $(\frac{m}{k}-1)$ recursive oblivious merger invocations, parties obtain the secret share of the k largest values among the input set. In summary, our secure k-priority construction requires $\left(\frac{1}{4}\log(k)+\frac{1}{2}\right)m\log(k)-\frac{1}{2}k\log(k)$ Compare-Swap operations.

The above discussion gives parties the secret shares of k largest values. To output their indexes, before running our k-priority protocol we attach the index with its value using the concatenation $||$. Namely, we use $(\ell+\lceil\log(m)\rceil)$-bit string to represent the input. The first ℓ bits to store the additive share $[\![v_i]\!]$ and the last $\lceil\log(m)\rceil$ bits to represent the index i. Therefore, within a group the oblivious sorting takes $\{[\![v_i]\!]||i,...,[\![v_{i+k-1}]\!]||(i+k-1)\}$ as input, use the shares $[\![v_j]\!], \forall j\in[i,i+k-1]$ for the secure comparison. The algorithm outputs the secret shares of the indexes, re-randomizes the shares of the values and swaps them if needed. The output of the modified oblivious sorting is $\{[\![v_{i_1}||i_1]\!],...,[\![v_{i_k}||i_k]\!]\}$ where the output values $\{v_{i_1},\dots,v_{i_k}\}\subset\{v_i,\dots,v_{i+k-1}\}$ are sorted. Similarly, we modify the oblivious merger structure to maintain the indexes. At the end of the protocol, parties obtain the secret share of the indexes of k largest values, which allows them jointly reconstruct the secret indexes.

Figure 4 presents our k-priority construction which security proof is given in the full version of the paper [25].

4.4 Putting All Together: MPCCache

We formally describe our semi-honest MPCCache construction in Fig. 3. From the preceding description, the cuckoo-simple hashing maps the same items into the same bin. Thus, for each bin $\#b$, if parties have the same $x_b^1\in B_1[b]$, they obtain the secret share of the sum of all corresponding associated values. Otherwise, they receive the secret share of zero (in practice, the sum of all parties' associated values for items in the intersection is not equal to zero). In our protocol, the equation $\bigoplus_{i=1}^{n}t_b^i=0$ determines whether the item x_b^1 is common. We choose the bit-length of the zero share to be $\lambda+\log(n)$ to ensure that the probability of the false positive event for this equation is overwhelming $(1-2^{-\lambda})$.

The second step of the online phase takes the shares from parties, and returns the indexes of k-priority common elements. Since k must be less than or equal to the intersection size, the obtained results will not contain an index whose value is equal to zero. In other words, the output of our protocol satisfies the MPCCache conditions since the identity is common and the sum of the values associated corresponding to this identity is k-largest.

The security of our decentralized MPCCache is based on OKVS and $\mathcal{F}_{\textsf{k-prior}}$ primitives. Its formal proof is given in the full version of the paper [25].

PARAMETERS:
- Number of parties n, set size m, and a k value
- An ideal oblivious sort $\mathcal{F}_{\mathsf{obv\text{-}sort}}$ and oblivious merge $\mathcal{F}_{\mathsf{obv\text{-}merge}}$ primitives described in Section 3.
- A truncation function trunc which returns first k elements in the list as $\{x_1, \dots, x_k\} \leftarrow \mathsf{trunc}(\{x_1, \dots, x_{2k}\})$

INPUT OF PARTY P_i: secret share values $S_i = \{[\![v_1]\!], ..., [\![v_m]\!]\}$

PROTOCOL:
1. Parties divide the input set S_i into $\frac{m}{k}$ groups, each has k items.
2. For each group $i \in [\frac{m}{k}]$ consisted of $\{[\![v_i]\!], ..., [\![v_{i+k-1}]\!]\}$ from party P_j, they jointly execute an oblivious sort $G[i] \leftarrow \mathcal{F}_{\mathsf{obv\text{-}sort}}(\{[\![v_i]\!]||i, ..., [\![v_{i+k-1}]\!]||(i+k-1)\})$, where $G[i] \overset{\mathsf{def}}{=} \{[\![v_{i_1}||i_1]\!], ..., [\![v_{i_k}||i_k]\!]\}$
3. Parties recursively invoke oblivious merges as follows. Assuming that $\frac{m}{k} = 2^d$

```
Procedure LevelMerge ($(G[0,\dots,d], d)$)
    if $d = 1$ then
        return $\{[\![v_{i_1}||i_1]\!], ..., [\![v_{i_k}||i_k]\!]\}$
    else
        $L = \mathsf{LevelMerge}(G[0,\dots,\frac{d}{2}-1], \frac{d}{2})$
        $R = \mathsf{LevelMerge}(G[\frac{d}{2},\dots,d-1], \frac{d}{2})$
        $M \leftarrow \mathcal{F}_{\mathsf{obv\text{-}merge}}(L, R)$
        where $M \overset{\mathsf{def}}{=} \{[\![v_{i_1}||i_1]\!], ..., [\![v_{i_k}||i_{2k}]\!]$
        $\{[\![v_{i_1}||i_1]\!], ..., [\![v_{i_k}||i_k]\!]\} \leftarrow \mathsf{trunc}(M)$
    end
end
```

4. Parties jointly reconstruct the share $\{[\![v_{i_1}||i_1]\!], ..., [\![v_{i_k}||i_k]\!]\}$, and output $\{i_1, \dots, i_k\}$.

Fig. 4. Our secure *k-priority* construction

5 Our Server-Aided MPCCache

In this section, we show an optimization to improve the efficiency of MPCCache. We assume that P_1 and P_2 are two non-colluding servers, and we call other parties as users. The optimized protocol consists of two phases. In the first one, each user interacts with the servers so that each server holds the same secret value, chosen by all users, for the common identifies that both servers and all users have. The servers also obtain the additive secret share of the sum of all the associated values corresponding to these common items. In a case that an identity x_j^e of the server $P_{e\in\{1,2\}}$ is not common, this server receives a random value. This phase can be considered as each user distributes a share of zero and a share of its associated value under a "common" condition. Note that, if even two servers collude they only learn the intersection items and nothing else, which provides a stronger security guarantee than the standard server-aided setting mentioned in the full version [25]. Our second phase involves only the servers' computation, which can be done by our 2-party decentralized MPCCache described in Sect. 4.4.

PARAMETERS:

- Set size m, a bit-length θ, security parameter λ, and n parties $P_{i\in[n]}$.
- A two-party decentralized $\mathsf{MPCCache}$, and an OKVS with Encode and Decode.

INPUT OF PARTY $P_{i\in[n]}$: A set of key-value pairs $S_i = \{(x_1^i, v_1^i), \dots, (x_m^i, v_m^i)\}$

PROTOCOL:

I. **Centralization**.
   1. Each user $P_{i\in[3,n]}$ chooses random $z_j^i \leftarrow \{0,1\}^{\lambda+\log(n)}$ and $s_j^i \leftarrow \{0,1\}^{\theta}$, and generates two sets $\Gamma^{e,i} = \{(x_j^i, z_j^i || w_j^{e,i})\}$, where $w_j^{1,i} \overset{\text{def}}{=} s_j^i$ and $w_j^{2,i} \overset{\text{def}}{=} v_j^i - s_j^i$.
   2. Each user $P_{i\in[3,n]}$ encodes $\Gamma^{e,i}$ as $\mathsf{Encode}(\Gamma^{e,i}) \rightarrow \mathcal{T}^{e,i}$ and sends $\mathcal{T}^{e,i}$ to $P_{e\in\{1,2\}}$ who computes $\mathsf{Decode}(x_j^e, \mathcal{T}^{e,i})$ and obtains $\hat{z}_j^{e,i} || \hat{w}_j^{e,i}$.
   3. For $j \in [m]$, each $P_{e\in\{1,2\}}$ computes $y_j^e \overset{\text{def}}{=} \bigoplus_{i=3}^n \hat{z}_j^{e,i}$ and $s_j^e \overset{\text{def}}{=} v_j^e + \sum_{i=3}^n \hat{z}_j^{e,i}$.

II. **Server-working**. Two servers $P_{e\in\{1,2\}}$ invoke an instance of $\mathsf{MPCCache}$ where P_e's input is a set $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ and learns k-$\mathsf{priority}$ common items.

Fig. 5. Our server-aided $\mathsf{MPCCache}$ construction.

More concretely, in the first phase, each user $P_{i\in[3,n]}$ chooses random $z_j^i \leftarrow \{0,1\}^{\lambda+\log(n)}$ and $s_j^i \leftarrow \{0,1\}^{\theta}$, and then defines $w_j^{1,i} \overset{\text{def}}{=} s_j^i$, and $w_j^{2,i} \overset{\text{def}}{=} v_j^i - s_j^i$. Next, $P_{i\in[3,n]}$ generates two sets of key-value points $\Gamma^{e,i} = \{(x_j^i, z_j^i || w_j^{e,i})\}, \forall e \in \{1,2\}$, computes $\mathcal{T}^{e,i} = \mathsf{Encode}(\Gamma^{e,i})$, and sends $\mathcal{T}^{e,i}$ to the server P_e. Let's $\hat{z}_j^{e,i} || \hat{w}_j^{e,i} \leftarrow \mathsf{Decode}(x_j^e, \mathcal{T}^{e,i})$ be an output of the OKVS decoding computed by $P_{e\in\{1,2\}}$. If two servers have the same item $x_k^1 = x_{k'}^2$ which is equal to the item x_j^i of the user P_i, we have $\hat{z}_k^{1,i} = \hat{z}_{k'}^{2,i} = z_j^i$ and $\hat{w}_k^{1,i} + \hat{w}_{k'}^{2,i} = v_j^i$ (since $\hat{w}_k^{1,i} = s_j^i$ and $\hat{w}_{k'}^{2,i} = v_j^i - s_j^i$). Each server $P_{e\in\{1,2\}}$ defines $y_j^e \overset{\text{def}}{=} \bigoplus_{i=3}^n \hat{z}_j^{e,i}$ as an XOR of all the obtained values $\hat{z}_j^{e,i}$ corresponding to each item $x_{j\in[m]}^e$. For two indices k and k', we have $y_k^1 = \bigoplus_{i=3}^n \hat{z}_j^{1,i} = \bigoplus_{i=3}^n \hat{z}_j^{2,i} = y_{k'}^2$ if all parties has $x_k^1 = x_{k'}^2$ in their set. This property allows servers obliviously determinate the common items (i.e., checking whether $y_k^1 = y_{k'}^2, \forall k, k' \in [m]$). Moreover, let $s_j^e \overset{\text{def}}{=} v_j^e + \sum_{i=3}^n \hat{w}_j^{e,i}$. For two indices k and k', s_k^1 and $s_{k'}^2$ are secret shares of the sum of the associated values for the common item $x_k^1 = x_{k'}^2$ In summary, after this first phase, each server $P_{e\in\{1,2\}}$ has a set of points $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ where $y_k^1 = y_{k'}^2$ if all parties have the same identity $x_k^1 = x_{k'}^2$, and $s_k^1 + s_{k'}^2$ is equal to the sum of the associated values of the common x_k^1. Therefore, we reduce the problem of n-party $\mathsf{MPCCache}$ to the problem of a two-party case where each server $P_{e\in\{1,2\}}$ has a set of points $\{(y_1^e, s_1^e), \dots, (y_m^e, s_m^e)\}$ and wants to learn the k-$\mathsf{priority}$ common items. We formally describe the optimized $\mathsf{MPCCache}$ protocol is in Fig. 5.

Recall that $y_j^e = \bigoplus_{i=3}^n \hat{z}_j^{e,i}, \forall e \in \{1,2\}, j \in [m]$. Let i be the highest index of a user $P_{i\in[3,n]}$ who did not have the identity x_k^1 in their input set. That user does not insert a pair $\{x_k^1, \texttt{something}\}$ to his set $\Gamma^{e,i}$ for the OKVS in Step (I.1). Thus, P_1 obtains a random $\hat{z}_k^{1,i}$ in Step (I.3). The protocol is correct except in the event of a false positive—i.e., $y_k^1 = y_{k'}^2$ for some x_k^1 not in the intersection.

By setting $\ell = \lambda + 2\log_2(n)$, a union bound shows that the probability of *any* item being erroneously included in the intersection is $2^{-\lambda}$.

The security proof of our server-aided MPCCache protocol is essentially similar to that of the decentralized protocol, which is presented in the full version [25].

Discussion. From our two-server-aided framework, our protocol can be extended to support a small set of servers (e.g., t servers, $t < n$). More precisely, in the centralization phase, each user $P_{i\in[t+1,n]}$ secretly shares their associated value $v^i_{j\in[m]}$ to the servers $P_{e\in[t]}$ via OKVS. Each server aggregates the share of the associated value corresponding to their item. The obtained results are forwarded to the server-working phase in which $P_{e\in[t]}$ jointly run MPCCache to learn k-priority common items. The main cost of our server-aided construction is dominated by the second phase. Hence, the performance of t-server-aided scheme is similar to that of decentralized MPCCache performed by t parties. We are interested in two-server aided architecture since we can take advantage of efficient two-party secure computation for the k-priority and GC. Moreover, the two-server setting is common in various cryptography schemes (e.g. private information retrieval [11], distributed point function [15], private database query [34]).

6 Implementation

We implement building blocks of MPCCache and do experiments on a single Linux machine that has Intel Core i7 1.88 GHz CPU and 16 GB RAM, where each party is implemented as a separate process. Computing cache sharing usually runs in the fast and low-latency edge network, especially with 5G technologies [1,3,13,37] as the servers of operators are typically placed closer to each other (e.g., in edge clouds in the same area such as New York City). Thus, we evaluate MPCCache over a simulated 10 Gbps network with 0.2 ms round-trip latency. We assume there is an authenticated secure channel between each pair of parties. Our MPCCache is very amenable to parallelization. Specifically, our algorithm can be parallelized at the level of bins. In our evaluation, however, we use a single thread to perform the computation between two parties.

All evaluations were performed with an identity and its associated value input length 128 bits and $\theta = 16$ bits, respectively, $\lambda = 40$, and $\kappa = 128$. We use OKVS code from [14], garbled circuit from [35]. To understand the scalability of our scheme, we evaluate it on the range of the number parties $n \in \{4, 6, 8, 16\}$. Note that the dataset size m of each party is expected to be not too large (e.g., billions). First, the potential of MPCCache is in 5G where each shared cache is deployed for a specific region. Second, each operator chooses only frequently-accessed files as an input to MPCCache because the benefit of caching less-accessed files is small. Therefore, we benchmark our MPCCache on the set size $m \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. To understand the performance effect of the k values discussed in Sect. 4.3, we use $k \in \{2^6, 2^7, 2^8, 2^9, 2^{10}\}$ in our k-priority experiments, and compare its performance to the most common oblivious sort protocol [30,35] which is based on Batcher's network (ref. Sect. 2).

Table 1. The total runtime (minute) and communication per item (KB) of our k-priority construction and the state-of-the-art oblivious sort, where m is the dataset size.

m	Running time					Communication				
	Ours k-priority				Sort [30, 35]	Ours k-priority				Sort [30, 35]
	$k = 2^7$	$k = 2^8$	$k = 2^9$	$k = 2^{10}$		$k = 2^7$	$k = 2^8$	$k = 2^9$	$k = 2^{10}$	
2^{12}	0.012	0.014	0.016	0.018	0.014	8.008	10.11	12.38	14.72	18.43
2^{14}	0.049	0.056	0.068	0.087	0.071	8.05	10.21	12.6	15.2	25.09
2^{16}	0.199	0.238	0.294	0.35	0.382	8.061	10.23	12.65	15.32	32.77
2^{18}	0.786	0.996	1.217	1.449	1.964	8.063	10.24	12.67	15.35	41.47
2^{20}	2.984	3.798	4.697	5.527	9.844	8.064	10.24	12.67	15.36	51.2

Table 2. The total runtime (minute) of our MPCCache constructions to find k-priority common items, where the number of parties n, each with dataset size m.

Parameters		Server-aided					Decentralized				
m	n	$k = 2^6$	$k = 2^7$	$k = 2^8$	$k = 2^9$	$k = 2^{10}$	$k = 2^6$	$k = 2^7$	$k = 2^8$	$k = 2^9$	$k = 2^{10}$
2^{12}	4	0.036	0.036	0.039	0.041	0.04	0.15	0.14	0.16	0.16	0.16
	6	0.036	0.036	0.039	0.041	0.04	0.23	0.22	0.24	0.23	0.27
	8	0.037	0.037	0.039	0.041	0.04	0.31	0.29	0.32	0.33	0.33
2^{16}	4	0.502	0.526	0.564	0.62	0.68	2.08	2.23	2.3	2.75	2.72
	6	0.502	0.531	0.569	0.625	0.68	3.09	3.06	3.71	3.65	3.96
	8	0.53	0.53	0.57	0.63	0.68	4.47	4.24	4.59	5.01	5.41
2^{20}	4	7.59	7.69	7.73	8.02	8.07	31.51	31.71	31.74	33.59	36.24
	6	7.7	7.92	7.81	8.1	8.17	46.07	46.35	46.37	46.69	46.96
	8	7.76	7.97	8.18	8.32	8.37	60.73	61.83	62.24	63.76	64.66

Table 3. The total runtime (minute) and communication cost per item (KB) of our server-aided MPCCache with $k = 2^8$ for the number of parties n, each with set size m.

#party n	Role	Running time (minute)					Communication (KB)				
		$m = 2^{12}$	$m = 2^{14}$	$m = 2^{16}$	$m = 2^{18}$	$m = 2^{20}$	$m = 2^{12}$	$m = 2^{14}$	2^{16}	$m = 2^{18}$	$m = 2^{20}$
4	User	0.002	0.003	0.088	0.324	1.202	0.58	0.66	0.73	0.81	0.88
	Server	0.039	0.146	0.564	2.089	7.732	24.47	26.34	28.06	29.74	31.41
6	User	0.002	0.004	0.093	0.342	1.271	1.17	1.32	1.46	1.61	1.76
	Server	0.039	0.147	0.569	2.1	7.813	24.77	26.67	28.43	30.14	31.85
8	User	0.002	0.004	0.095	0.35	1.291	1.75	1.97	2.19	2.42	2.64
	Server	0.039	0.147	0.571	2.12	7.781	25.06	27	28.79	30.54	32.28
16	User	0.02	0.058	0.24	0.912	3.374	4.09	4.61	5.12	5.64	6.15
	Server	0.047	0.167	0.598	2.155	7.833	26.23	28.32	30.26	32.15	34.04

6.1 k-priority Performance

Our k-priority requires $\left(\frac{1}{4}\log(k) + \frac{1}{2}\right)m\log(k) - \frac{1}{2}k\log(k)$ Compare-Swap instances. We use GC [5, 36] to perform secure comparisons. Table 1 presents the running time and communication cost of our k-priority for the different k values. The cost is measured in KB per item as we would like to show an improved performance factor of our proposed protocol compared to the state-of-the-art oblivious sort as well as a performance change when increasing k. Thus, for

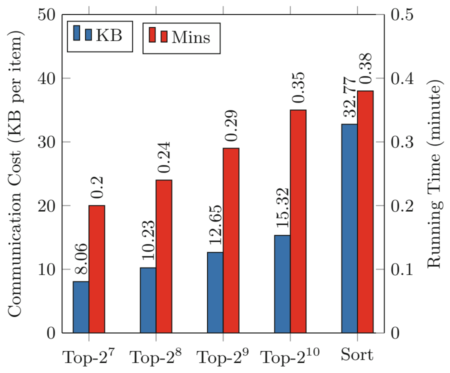

Fig. 6. The total running time (red bar) in minute and communication cost (blue bar) per item in KB of our k-priority and oblivious sort for Top-k and data set size $m = 2^{16}$. (Color figure online)

$m = 2^{18}$ and $k = 2^7$, our approach shows 5.15× and 2.5× improvements in terms of communication and computational costs, respectively.

To see more clearly the performance change for different k values, we present the performance of our k-priority protocol using a bar chart in Fig. 6, and show that there is a minor change in the running time when increasing k.

6.2 MPCCache Performance

Table 2 presents the total running time for the decentralized and server-aided MPCCache. The main difference between these constructions is in the steps of GC equality checks and k-priority. While the decentralized scheme requires all participants to jointly compute these steps, in the server-aided framework only two specific servers perform the computation. Thus, the former model is expensive than the latter one but provides a stronger security guarantee where any subset of corrupted parties learns nothing about the dataset of honest parties.

The numbers reported in Table 2 are for an end-to-end server-aided MPCCache execution, which includes the user's waiting time for the servers's computation. As discussed Sect. 5, the server-aided protocol is asymmetric with respect to the servers $P_{e\in\{1,2\}}$ and other users. Table 3 presents the performance of different roles of the participants. Because the user only distributes its dataset to two servers in the centralization phase, his workload is very light. The performance of our server-aided MPCCache on the user's side does not depend much on the number of parties due to the parallelizability with a separate secure channel between user and server. The server's work is heavy due to equality checks and k-priority. Table 3 shows that our protocol scales to a large number of parties.

6.3 Comparison with Prior Work

We compare our protocols with recent related works [7,31]. One can extend MPCircuits [31] to address the multi-party cooperative cache sharing problem by following similar steps of MPCCache: the first phase is to compute the secret share of the intersection. The second phase uses generic MPC protocols or our k-priority to compute the top-k function on the obtained results. Recall that MPCircuits only allows to compute secret-shared intersection items themselves. It is based on a binary tree structure as [31] observed that the set intersection of n sets can be expressed as a consecutive set intersection of two sets until reaching the final result. Therefore, the intersection of two sets is computed at each node of the tree, and the final intersection of all sets is computed at the root of the tree. Using three operations as sort, merge, and compare, the complexity of their garbled circuit is $O(n^2 m\ell \log(m)^2)$ where ℓ is the bit-length of the element identity. To keep track θ-bit associated value of the identity, the MPCircuits-based solution requires a complexity of $O(n^2 m(\ell + \theta) \log^2(m))$. In contrast, with the lightweight OKVS, our solution requires only a single equality comparison per bin. Thus, the complexity of our circuit is $O(nm(|z| + \theta))$, where z is a bit-length of the zero share which is equal to $\min(\ell, \lambda + \log(n))$. It is easy to see that the first phase of our solution is about $n \log^2(m)\times$ better than that of MPCircuit-based approach. For example, with $n = 8$ and $m = 2^{20}$ our solution shows about an $3,200\times$ improvement.

To hide the intersection set size, the output of the MPCircuits-based computation at the root of the tree consists of mn secret shares of all intersection and non-intersection items. As a result, the second phase of the baseline solution takes mn secret shares as an input of each party. On the other hand, our MPCCache only takes $\beta = 1.27m$ secret shares, each per bin.

A concurrent and independent work [7] is designed for a generic circuit-PSI which only supports an honest majority (e.g., the number of colluding parties is up to $t < n/2$). Their protocol is similar to MPCCache and consists of two main phases. However, the first phase of [7] requires expensive steps (e.g., multiplication on secret-shared values) to compute the shares of intersection (Step 6 &7, [7, Figure 6]). Moreover, each participant (e.g. client) of [7] has a computation/communication complexity $O(nm)$ and requires to participate in the mostly full computation process. In contrast, in our server-aided protocol, the client does not involve in the entire MPCCache computation process, thus, has commutation/communication complexity $O(tm)$ which is independent of n. According to [7, Table 4] for $m = 2^{20}, n = 5, t = 2$ their client expects to finish the first phase in 25.48 s while ours requires only 13.02 s, an $1.96\times$ improvement[1]. The improvement factor is higher when the ratio n/t is larger.

For the second phase, [7] is not customized for the top-K computation. Based on the theoretical analysis in Sect. 6.1 and numerical experiment in Sect. 4.3, we expect that the second phase of MPCCache is about 1.7–3.3$\times$ faster than [7].

[1] [7]'s implementation is not yet publicly available. Its benchmark machine is stronger than ours, which is in favor of their protocol.

Acknowledgements. The second author is partially supported by NSF awards #2101052, and #2115075.

References

1. AT&T Edge Cloud (AEC) - White Paper (2017). https://about.att.com/ecms/dam/innovationdocs/Edge_Compute_White_Paper%20FINAL2.pdf
2. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 805–817. ACM Press, October 2016
3. Bastug, E., Bennis, M., Debbah, M.: Living on the edge: the role of proactive caching in 5G wireless networks. IEEE Commun. Mag. **52**(8), 82–89 (2014)
4. Batcher, K.E.: Sorting networks and their applications. In: Spring Joint Computer Conference (1968)
5. Beaver, D., Micali, S., Rogaway, P.: The round complexity of secure protocols. In: STOC (1990)
6. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the internet. In: CCS 2016 (2016)
7. Chandran, N., Dasgupta, N., Gupta, D., Obbattu, S.L.B., Sekar, S., Shah, A.: Efficient linear multiparty PSI and extensions to circuit/quorum PSI. ePrint (2021)
8. Chandran, N., Gupta, D., Shah, A.: Circuit-PSI with linear complexity via relaxed batch OPPRF. ePrint (2021)
9. Chen, H., Chillotti, I., Dong, Y., Poburinnaya, O., Razenshteyn, I., Riazi, M.S.: SANNS: scaling up secure approximate k-nearest neighbors search. In: USENIX Security (2020)
10. Cheon, J.H., Jarecki, S., Seo, J.H.: Multi-party privacy-preserving set intersection with quasi-linear complexity. IEICE Trans. **95**(8), 1366–1378 (2012)
11. Corrigan-Gibbs, H., Kogan, D.: Private information retrieval with sublinear online time. In: Canteaut, A., Ishai, Y. (eds.) EUROCRYPT 2020. LNCS, vol. 12105, pp. 44–75. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45721-1_3
12. Demmler, D., Rindal, P., Rosulek, M., Trieu, N.: PIR-PSI: scaling private contact discovery. In: Privacy Enhancing Technologies Symposium (PETS) (2018)
13. ETSI: Multi-access edge computing (2019). https://www.etsi.org/technologies/multi-access-edge-computing
14. Garimella, G., Pinkas, B., Rosulek, M., Trieu, N., Yanai, A.: Oblivious key-value stores and amplification for private set intersection. In: Malkin, T., Peikert, C. (eds.) CRYPTO 2021. LNCS, vol. 12826, pp. 395–425. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-84245-1_14
15. Gilboa, N., Ishai, Y.: Distributed point functions and their applications. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 640–658. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-55220-5_35
16. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: Aho, A. (edr.) 19th ACM STOC, pp. 218–229. ACM Press, May 1987
17. Hazay, C., Venkitasubramaniam, M.: Scalable multi-party private set-intersection. In: Fehr, S. (ed.) PKC 2017. LNCS, vol. 10174, pp. 175–203. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54365-8_8
18. Jónsson, K.V., Kreitz, G., Uddin, M.: Secure multi-party sorting and applications. ePrint (2011)

19. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) ACM CCS 2016, pp. 818–829. ACM Press, October 2016
20. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: Thuraisingham, B.M., Evans, D., Malkin, T., Xu, D. (eds.) ACM CCS 2017, pp. 1257–1272. ACM Press, October 2017
21. Lepoint, T., Patel, S., Raykova, M., Seth, K., Trieu, N.: Private join and compute from PIR with default. In: Tibouchi, M., Wang, H. (eds.) ASIACRYPT 2021. LNCS, vol. 13091, pp. 605–634. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92075-3_21
22. Mohassel, P., Rindal, P.: ABY^3: a mixed protocol framework for machine learning. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) ACM CCS 2018, pp. 35–52. ACM Press, October 2018
23. Neugebauer, G., Meyer, U., Wetzel, S.: SMC-muse: a framework for secure multi-party computation on multisets. In: INFORMATIK 2013 - Informatik angepasst an Mensch, Organisation und Umwelt (2013)
24. Nevo, O., Trieu, N., Yanai, A.: Simple, fast malicious multiparty private set intersection. In: ACM Conference on Computer and Communications Security (CCS) (2021)
25. Nguyen, D.T., Trieu, N.: MPCCache: privacy-preserving multi-party cooperative cache sharing at the edge. Cryptology ePrint Archive, Report 2021/317 (2021). https://ia.cr/2021/317
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
27. Paschos, G.S., Iosifidis, G., Tao, M., Towsley, D., Caire, G.: The role of caching in future communication systems and networks. IEEE J. Sel. Areas Commun. **36**(6), 1111–1125 (2018)
28. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: private set intersection using permutation-based hashing. In: Jung, J., Holz, T. (eds.) USENIX Security 2015, pp. 515–530. USENIX Association, August 2015
29. Pinkas, B., Schneider, T., Tkachenko, O., Yanai, A.: Efficient circuit-based PSI with linear communication. In: Ishai, Y., Rijmen, V. (eds.) EUROCRYPT 2019. LNCS, vol. 11478, pp. 122–153. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17659-4_5
30. Poddar, R., Kalra, S., Yanai, A., Deng, R., Popa, R.A., Hellerstein, J.M.: Senate: a maliciously-secure MPC platform for collaborative analytics. In: USENIX (2021)
31. Riazi, M.S., Javaheripi, M., Hussain, S.U., Koushanfar, F.: MPCircuits: optimized circuit generation for secure multi-party computation. In: HOST, pp. 198–207 (2019)
32. Rindal, P., Schoppmann, P.: VOLE-PSI: Fast OPRF and circuit-PSI from vector-OLE. ePrint (2021)
33. Sang, Y., Shen, H.: Privacy preserving set intersection based on bilinear groups. In: ACSC 2008 (2008)
34. Wang, F., Yun, C., Goldwasser, S., Vaikuntanathan, V., Zaharia, M.: Splinter: practical private queries on public data. In: NSDI (2017)
35. Wang, X., Malozemoff, A.J., Katz, J.: EMP-toolkit: efficient MultiParty computation toolkit (2016). https://github.com/emp-toolkit
36. Yao, A.C.-C.: How to generate and exchange secrets (extended abstract). In: 27th FOCS, pp. 162–167. IEEE Computer Society Press, October 1986

37. Yao, J., Han, T., Ansari, N.: On mobile edge caching. IEEE Commun. Surv. Tutor. **21**(3), 2525–2553 (2019)
38. Zhang, K., Leng, S., He, Y., Maharjan, S., Zhang, Y.: Cooperative content caching in 5G networks with mobile edge computing. IEEE Wirel. Commun. **25**(3), 80–87 (2018)